May 1980

Multiprocessing Extensions for the RMX/80™ Real-Time Executive

Peter Andersen
OEM Microcomputer Systems Applications

Using the RMX/80™ Nucleus in Multiprocessor Applications

Contents

I. INTRODUCTION

As computer systems become more complex, a recurring need occurs to use multiple processing units. Multiprocessing can, in many applications, increase throughput, enhance reliability, or create a more modular design approach. Intel's single board computer family provides a convenient tool for creating a multiprocessing environment when combined with the Multibus system bus architecture. The use of the Intel RMX/80 Real-Time Multitasking Executive simplifies the creation of modular designed system architectures. It also allows more efficient use of the processor's capabilities by sharing them between several independent tasks.

This application note provides insight as to the techniques which can be used to create a multiprocessing system which is fully compatible with the use of Intel's RMX/80 multitasking executive. Both hardware and software capabilities of the Intel single board computers are discussed. For example, an explanation of both serial and parallel priority resolution techniques for creating a multimaster system are included. The techniques for using the bus lock features to provide mutual exclusion of resources and for creating both hardware and software semaphores is also a part of the application note. Many of the multiprocessing principles are combined to create the capability of transferring messages between tasks operating on different processor boards, thus creating a combined multiprocessor/multitasking operating system.

A potentially complex application is examined and the benefits gained by using a multiprocessing approach are discussed. The various requirements defined from the application are used to demonstrate the available features of the Intel product line as applied to multiprocessing.

Extensions of the features are developed where necessary to create a complete solution to the design exercise. For example, a capability is provided for a flexible operator interface to the system which provides certain global functions to the multiple processors. This capability, called the Common System Module (CSM), includes the required drivers for disk drives and for an operator's CRT terminal. The operator interface includes a BASIC interpreter capability.

Finally, the application is modularized and the techniques which make the control solution easily implemented using multiprocessing techniques are demonstrated.

It is assumed that the reader has a fundamental knowledge of multitasking operating system concepts and is familiar with the use of PL/M-80 as a language on Intel's single board computers. In addition, a working knowledge of BASIC is assumed. The reader not having these skills should refer to the documents referenced in the Related Publications section of this application note.

Multiprocessing Strengths

The most common use of multiprocessing is to offload a main processor of low level duties in order to increase system throughput. Significant examples of this technique are the use of Intel UPI devices such as the 8741A and the use of intelligent slave boards such as the Intel iSBC 544 Intelligent Communications Controller or the iSBC 569 Intelligent Digital Controller. The techniques which are developed in this application note do not apply to these multiprocessor implementations because they have been extensively covered in other application notes.

Processors can be offloaded of other than low level functions in complex systems. Just as a real-time application can be divided into functional tasks, these tasks can be grouped according to function. Each function or class of tasks can be assigned to a single board computer. The system throughput can thus be increased since tasks can now run concurrently and, using the software procedures developed in this application note, can communicate among themselves as though they were operating on the same processor board. Single board computers operating in this manner are known as multimaster computers.

A second instance where multiprocessing can be considered in a system design is in those cases where the operational integrity can be enhanced by having more than one processor. In these cases, the application may require that no single component failure can completely cause the system to fail. Multiprocessing provides this feature by allowing control of those operations associated with a processor board to continue to function even when another board has failed. In many cases, provision can be made to have the remaining board or boards take over some or all of the functions which were associated with the failed board. Even though this would result in slower throughput, critical functions would still be performed by the computer system.

Another benefit is the modular system design and expansion capabilities which can be realized by using multiprocessing in industrial control applications. Here, it has been found that process definitions and operations tend to be rather dynamic. As products are improved or new ones added to the operation, the control system must be capable of being updated to conform to the new product flow. It is common to have an added requirement that the changes in a process must be made with no impact on other processes being controlled by the same system. Using a multimaster board for each process allows this operation to be easily implemented.

Process Example

To emphasize the advantages which can be gained using multiprocessing, consider the chemical production facility whose process flow is shown in Figure 1. This product flow chart is typical of many found in the chemical industry where a product is manufactured from several

raw ingredients which are purchased. This type of product lends itself well to automation. Consider, then, the approach to designing a control system which will operate in the facility which has been defined.

The first design goal is to define global system tasks which need to be performed by the system. Examination of the flow diagram in Figure 1 might lead one to define the system tasks as:

1) Inventory control
2) Quality control
3) Production scheduling
4) Weighing ingredients
5) Mixing ingredients
6) Drying and forming product
7) Packaging

The choice just made certainly is not the only valid possibility but it should be adequate to allow the design discussion to continue.

The system tasks listed can be grouped according to supervisory and control tasks. Items 1 through 3 fall into the supervisory category and items 4 through 7 belong to control category. This breakdown also groups the items according to the type of programs which will be required to implement the functions.

The grouping of system functions into categories suggests that a multimaster approach is a good design path to create a system solution. Five multimaster boards are suggested, one for the supervisor and four boards which provide the control capabilities. This functional grouping is seen in Figure 2.

The examination of the product flow and system interactions shown in Figure 1 indicate that extensive communications are required between the various boards of the system. In addition, several tasks are required on each multimaster board and communications are required between these tasks.

Intel's RMX/80 Real-Time Multitasking Executive provides a simple method of handling the on-board communication operations by supporting the transmission and receipt of messages for data communication and synchronization. After a discussion of the hardware features which support multiprocessing, the application note will deal with the extension of the RMX/80 nucleus to support the transmission of messages between tasks which reside on different single board computers.

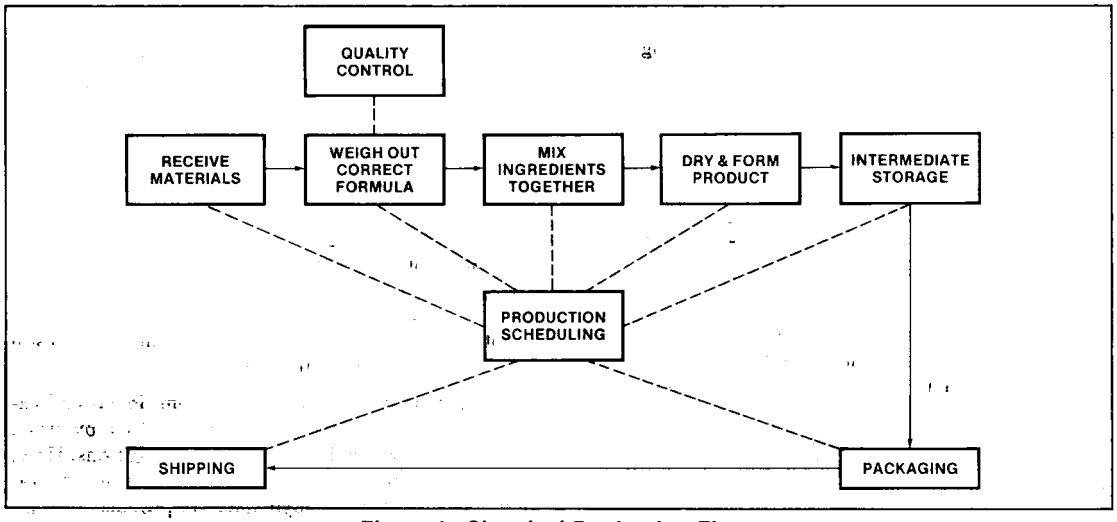

Figure 1. Chemical Production Flow

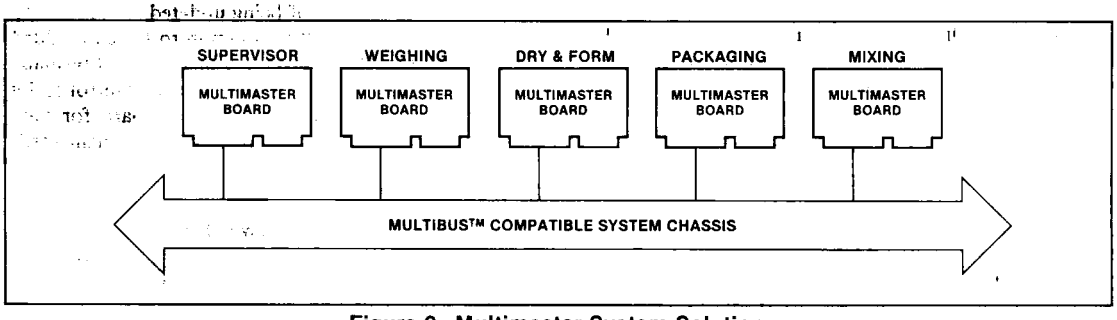

Figure 2. Multimaster System Solution

II. MULTIBUS™ MULTITASKING OPERATIONS

The interactions of the various multimaster board functions occur via the medium of the Multibus system bus. This bus structure is designed to easily support the use of more than one master processor in a system. Subsequent paragraphs explain in some detail the features of the Multibus system bus which allow multimaster operations when used with multimaster compatible single board computers.

Bus Priority Resolution Lines

The Multibus system bus uses seven control lines to support the bus priority resolution techniques. Before continuing with a discussion of these techniques, the reader should be provided with a definition of these lines and their functions. These lines are defined in subsequent paragraphs. One line is used for the system clock. It is:

BCLK/ Bus Clock; the negative edge of BCLK/ is used to synchronize bus priority resolution circuits. BCLK/ is asynchronous to the CPU clock. It has a 100 ns minimum period and a 35 to 65 percent duty cycle.

All Intel single board computers which are capable of being used as a bus master can provide this clock signal. In a multiple processor application, it is necessary to select only one board which is to actually supply this signal to the system bus. On all other processor boards, this signal must be disabled by removing the jumpers as indicated in the appropriate Hardware Reference Manual (HRM).

Several bus signals are dedicated to providing information as to the status of the bus to processors which are attempting to obtain control for their own data transfer. These are defined to be:

BPRN/ Bus Priority In Signal; indicates to a requesting bus master board that no higher priority board is requesting use of the system bus. BPRN/ is synchronized with BCLK/. The signal is not bused on the backplane and must be generated and connected by the user as will be seen.

BUSY/ Bus Busy Signal; an open collector line driven by the current bus master to indicate that the bus is currently in use. BUSY/ prevents all other potential masters from gaining control of the bus. BUSY/ is synchronized with BCLK/.

CBRQ/ Common Bus Request; an open-collector line which is driven by all potential bus masters and is used to inform the current bus master that another device wishes to use the bus. If CBRQ/ is high, it indicates to the current bus master that no other master is requesting the bus, and therefore, the present master can retain the bus control. This saves the bus exchange overhead which would be required to release and again re-acquire control of the bus.

The timing relationship of these signals can be seen in Figure 3. Note that some type of bus request signal is generated by the master wishing to assert control of the system bus. Through some type of logic, a determination must be made as to the effect that the requesting master has the highest priority. If so, the logic informs the requesting master by setting the BPRN/ signal low.

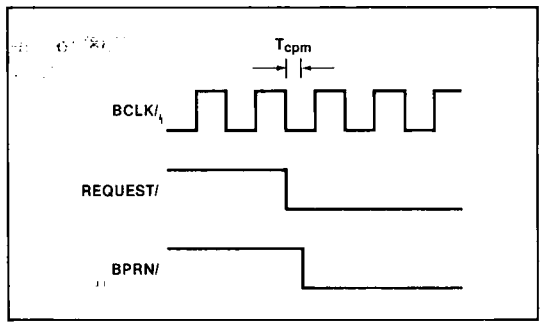

Figure 3. Bus Master Request Timing

Two signals are generated by single board products designed to be bus masters. These signals are used to generate the request signal. The request lines provided are known as:

BPRO/ Bus Priority Out Signal; used with serial bus priority resolution schemes. BPRO/ is passed to the BPRN/ input of the master module with the next lower priority. BPRO/ is synchronized with BCLK/ and is not bused on the backplane.

BREQ/ Bus Request Signal; used with a parallel bus priority network to indicate that a particular master board requires the use of the bus for one or more data transfers. BREQ/ is synchronized with BCLK/. It is not bused on the backplane.

The most easily implemented technique for supporting bus priority resolution is the serial or the daisy chain method. In this method, each board examines its BPRN/ line. If the line is low and the master desires to use the system bus, it may do so. Internal board logic then places the BPRO/ line high to inhibit boards along the daisy chain with lower priority from gaining control. Any board in the chain with its BPRN/ line high or which is requesting priority will set the BPRO/ line high. The wiring technique and logic of the serial priority resolution scheme is shown in Figure 4. As long as the worst case propagation delay time between the highest and the lowest potential bus master does not exceed 30 ns, the technique provides a simple and effective method for resolving bus contention problems. Because of logic delays on iSBC boards, it has been found that the serial technique can only be used for a maximum of three bus masters when using a 100 ns clock rate. If

more masters are desired, either the clock must be slowed or the priority resolution technique must be modified. Thus, the BREQ/ line is used to support the second or parallel priority scheme.

In a parallel priority network, all requests from bus masters are supported in parallel logic. Each master desiring use of the system bus places its BREQ/ line to a logical low condition. A parallel priority encoder chip such as a 74148 can be used to indicate the requestor having the highest priority. The output of the encoder can next be fed into a decoder network (74S138) to generate a unique enable signal back to the processor board. These signals are fed back to the requesting boards as the BPRN/ control line. The logic to perform this parallel resolution is not a part of current Intel Multibus backplanes, so it must, when required, be constructed and supplied by the user. Figure 5 shows an implementation which can be used to support a twelve-slot cardcage such as can be obtained using the Intel iCS-80 chassis. Since all bus priority requests are handled using parallel logic, additional potential bus masters may be added without adding to the signal delay time. Thus, the number of requests which may be resolved is limited only by the number of available card slots on the Multibus system bus.

Resource Contention

Even though the bus priority resolution networks described above provide against two or more bus masters taking control of the bus at the same time, they do not prevent one processor from examining and modifying data at the same time that another is operating on this data (this is true because data is only transferred in 8 or 16-bit blocks and the bus may be used by another master between transfers). Some technique for inhibiting the contention of resources must be provided if true multiprocessing is to be performed. Three candidates can be considered to support this exclusion process. The techniques required to support each are explored in subsequent paragraphs.

BUS LOCK

Single board computers provide the capability of locking the system bus from access by other bus masters. This feature is provided by means of a bus lockout flip-flop which may be set or reset by writing to a dedicated I/O port on each board. This flip-flop has the effect of keeping the BUSY/ active which keeps the master in control of the bus. In this way, a master assures that it has exclusive control of a bus common resource. In fact, it has exclusive control of all bus resources even though

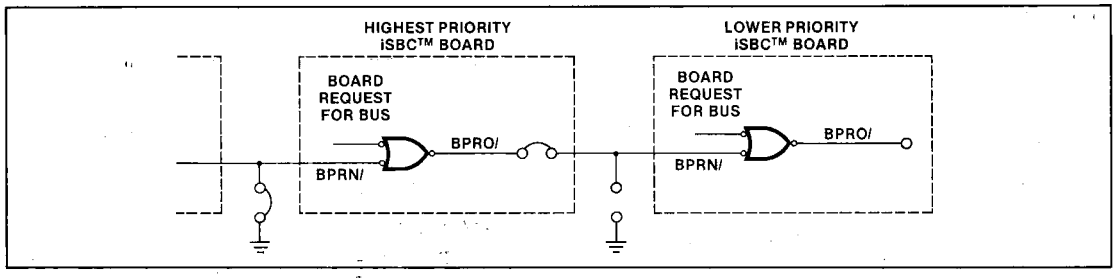

Figure 4. Serial Priority Implementation

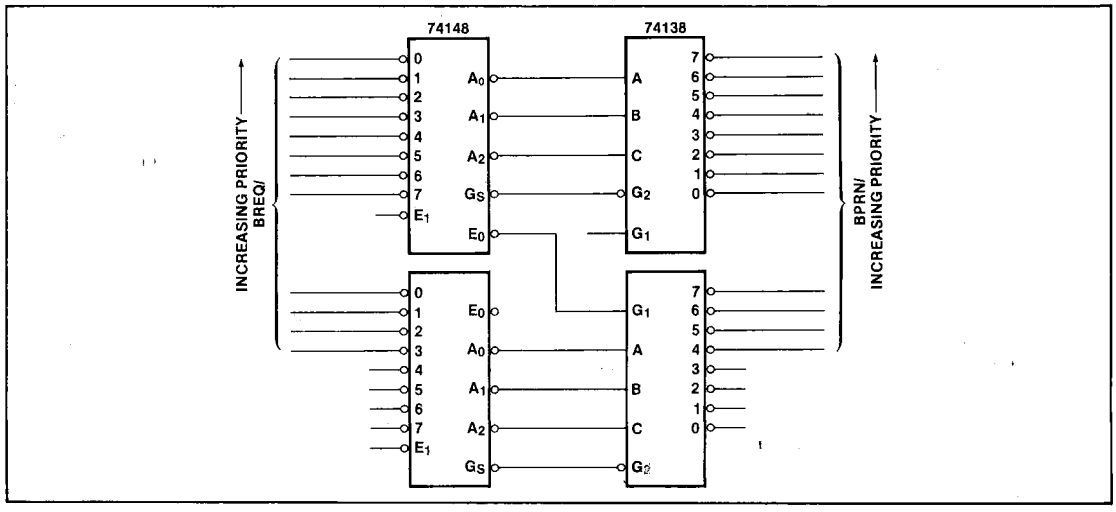

Figure 5. Parallel Priority Implementation

it may be using but one of them. This drawback prohibits other masters from using devices on the bus which are not in contention as well as those that are. In many systems, this potential delay can prohibit the use of bus locks in all but a few special applications.

SEMAPHORES

Mutual exclusion can easily be obtained by using the concept of a semaphore. A semaphore may be thought of as a type of traffic signal which can be used to prohibit access to certain roads or resources. When such a semaphore technique is used with microprocessors, it is referred to as a test-and-set operation. This means that the act of testing the semaphore not only returns the current status but also forces the semaphore to the "on" condition. If the returned status from the test indicated a previously cleared condition of the semaphore, the resource associated with it is considered available for exclusive use. Since the semaphore is now set, all other masters testing it will find a busy condition and must wait until the user processor clears the semaphore assocaited with the resource. Each time a processor completes its operations on a shared resource, it clears the semaphore by writing a zero to it.

In practice, two types of semaphores are commonly found, the hardware semaphore and the software semaphore. The use of each is identical and follows the general concepts outlined in the preceding paragraphs.

Hardware Semaphores

Hardware semaphores are those which are implemented using physical hardware. Logic components are used in circuits which must be designed and constructed by the user. Currently, no Intel boards provide for this type of semaphore implementation. Hardware semaphores have the advantage over a software implementation of being faster and requiring less system bus time. The most common designs consider the semaphore as an I/O port although memory mapped designs are certainly feasible.

Hardware semaphores are conceptually thought of as a D-type flip-flop. A simplified diagram of a typical semaphore is seen in Figure 6(a). Examining the timing relationships of Figure 6(b), it is seen that, if an initial state of a clear semaphore is assumed, when a master processor reads the I/O port, a data value of "zero" will be returned. The hardware immediately sets the flip-flop to a logical "one" on the trailing edge of the read signal. Any subsequent reads will find the semaphore "set". When the user is finished with the resource, it can clear the flip-flop by performing a "write" operation to the I/O port.

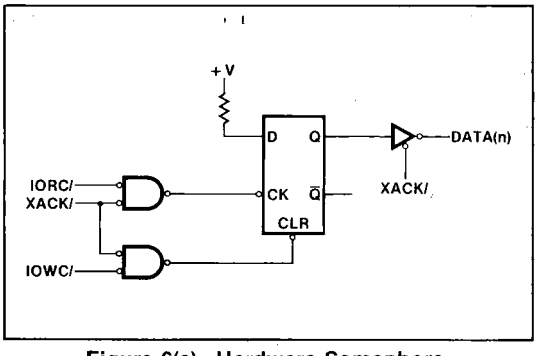

Figure 6(a). Hardware Semaphore

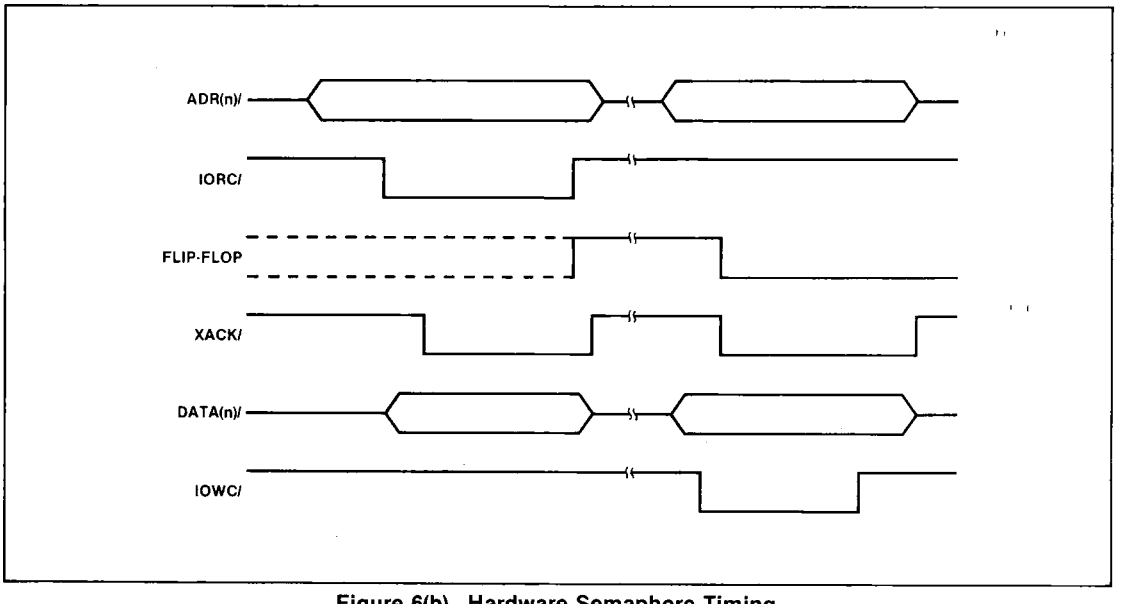

Figure 6(b). Hardware Semaphore Timing

The user can prevent contention of a system resource by incorporating a "wait for semaphore clear" into his program. For example, if a resource is protected by a semaphore at I/O port 35H, the following PL/M code could be used to protect the resource:

```
/* WAIT FOR SEMAPHORE TO CLEAR, LOCK IT */
   DO WHILE INPUT (035H)  < >  0; END;
/* PERFORM OPERATIONS ON RESOURCE */
   .
   .
   .
/* CLEAR SEMAPHORE TO ENABLE RESOURCE */
   OUTPUT (035H) = 0;
```

The hardware semaphore is seen to be an effective method of incorporating mutual exclusion in a multiprocessing application. Its major asset is that it minimizes system bus overhead and assures maximum bus throughput by higher priority master devices. Its weakness is the requirement for the user to provide hardware to implement the semaphores. In many cases, the bus overhead incurred by continuous sampling of the semaphore when it is busy can be eliminated by providing a system bus interrupt each time a semaphore is cleared. If this technique is used, a processor desiring a busy resource could enable the appropriate interrupt level, then wait until the semaphore is cleared by the current user.

Software Semaphores

The concept of software semaphores has long been used in programming applications. A byte of memory is reserved for the semaphore or flag and this byte is then tested by software programs to convey data. This implementation is not valid for multiprocessor designs since there is nothing to prevent a second processor from testing and finding a semaphore clear while the first is attempting to set it. To be effective in multiprocessor systems, the software semaphore must exhibit the characteristics of the "test-and-set" hardware implementation.

Many Intel single board computers such as the iSBC 86/12A board incorporate a special instruction prefix which provides the test-and-set feature. This prefix, the LOCK, causes the processor board to maintain control of the system bus while the byte or word operand is tested and set. PL/M-86 incorporates a special instruction which implements the software semaphore. The use of this instruction is:

```
oldvalue = LOCK SET (memory address, newvalue)
```

This instruction causes the processor board to lock the bus and place the new value into the specified memory location. The old value is returned to the calling program. Only then will the bus be unlocked. An example demonstrates how this provides mutual exclusion of a resource.

Assume that a software semaphore, LOCK$BYTE has been declared. A value of 1 indicates that the resource is being used and is not available. A value of 0 indicates that the common resource is available to a task. If the function reference LOCKSET(@LOCK$BYTE, 1) is executed, the value of 1 will be assigned to LOCK$BYTE. Furthermore, the old value of the semaphore will be returned. If a value other than 0 is returned, the statement must be repeated as the resource is being used by another task. When a value of 0 is returned as the old value, the resource has been dedicated to the calling task. When the task is finished with the shared resource, a zero must be written to the semaphore byte.

A similar technique can be used with 8-bit single board computers. For example, consider the iSBC 80/30 board. Provision has been made to provide a bus lock condition when the CPU SOD (the SOD is a output data line unique to the 8085 processor which is normally intended to be used as a serial output line) is active. A task desiring to gain exclusive access to a common resource, can lock the bus, then test and set the semaphore flag; finally, the bus can be freed by clearing the SOD signal. An example will make this more clear.

Consider the same semaphore which was used in the PL/M-86 discussion. A task executing on an iSBC 80/30 board and desiring to use the protected resource would have code similar to the following:

```
/* WAIT FOR RESOURCE AVAILABLE */
   OLDVALUE = 1;
   DO WHILE OLDVALUE = 1;
     CALL S$MASK (0C0H);          /* lock bus       */
     OLD VALUE = LOCK$BYTE; /* get old value  */
     LOCK$BYTE = 1;               /* set semaphore */
     CALL S$MASK (040H);          /* unlock bus     */
   END;

/* Perform necessary operations on protected resource */
   .
   .
   .

/* Unlock resource for other tasks to use */
     LOCK$BYTE = 0;               /* clear semaphore */
```

Other boards use a dedicated I/O port to control the bus lock feature. For example, the iSBC 80/24 single board computer has dedicated port D5 to control the lock. Writing a 1 to this port will lock the bus and writing a 0 will unlock it. With this exception, the technique shown for the iSBC 80/30 board will work when using the iSBC 80/24 processor board in a multiprocessing application.

III. MULTIPROCESSOR COMMUNICATIONS

For efficient operation of a system solution which uses more than one processor, a means must be found to transmit data between the system bus master boards. The data, so transferred, might be used to pass param-

eters, to move data strings, or to synchronize events on the various boards. While it is not the intent of this application note to fully explain all possible techniques, it does provide some insight into the most common multiprocessor data transfer methods.

After a short discussion of various transfer mechanisms, the techniques used in the RMX/80 nucleus are described. The descriptions provide a basis for the development of an extension to the nucleus which allows tasks to reside on multiple boards in a multimaster environment.

The most straightforward technique to communicate between processors is the use of flags or semaphores to synchronize operations. The procedures used in this type of data transfer have been explained in previous paragraphs. As will be seen, this technique will be later combined with more complex methods to synchronize the transfer of data.

A second type of communications which may be used involves creating and maintaining data queues. This method was created to support a special case of multiprocessing, the use of intelligent slaves. The creation and support of data queues has been thoroughly detailed in an Intel application note, *AP-53, Using the iSBC 544 Intelligent Communications Controller*. It should be noted that, even though created for intelligent slaves, the technique is certainly applicable to the general case of multiprocessor communications.

The generalized data queue is designed to primarily deal with the movement of data strings between processors. The queue provides independent operation for each of the two involved processors, which may operate upon the data contained within the queue independently. The queue handlers, however, must interact since the queue pointers must never be allowed to pass each other or invalid data would be handled. This implies that mutual exclusion must be applied to the pointers to quarantee their integrity. If this is done, the queue can readily be applied to the special cases where string data transfers must take place between various processor boards.

Multiprocessor systems infer that the application breaks down into functionally independent blocks or "tasks". Frequently, these global tasks will most often be further subdivided into smaller on-board or local tasks. These systems will frequently use real-time executive operating systems.

Communications between tasks can involve the transfer of many types of information. In some cases, data strings must be moved. In others, the transfer may be only used for synchronization of events. Intel's RMX/80 multitasking operating system supports the movement of data (messages) through the use of exchanges. In this implementation, actual messages are not moved in memory, but instead, a pointer is generated to the message area and this variable is passed between tasks. Before proceeding with the development of a multiprocessor message driver, some time must be taken to assure that the message/exchange relationships are fully understood by the reader.

Messages

A message is a collection of data that one task sends to another task. It may be thought of in the context of a letter which is to be sent from one person (task) to another. Like a letter, the purpose of the message is to convey information or to request a service. Messages used in RMX/80 systems conform to a standard format similar to conventional letter connotations. It contains a heading and a variable length data area. The heading can be from 5 to 9 bytes in length and corresponds to the format shown in Figure 7.

The LINK PORTION of the heading is used by the RMX/80 nucleus to keep track of other messages which are addressed to the same place (exchanges). If no other messages are present, the field will be set to zero.

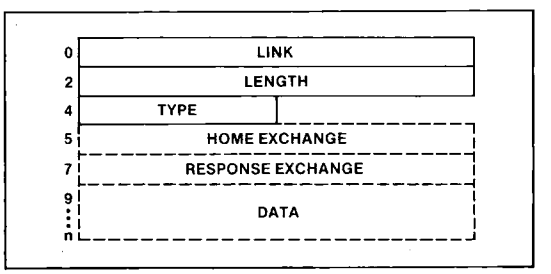

Figure 7. RMX/80™ Message Format

Exchanges

The exchange can be thought of as a mailbox into which messages are placed. In reality, only the location pointer of the message is placed into the exchange. Each exchange is defined by an Exchange Descriptor area of RAM memory. The format of an Exchange Descriptor is shown in Figure 8. The purposes of each field will be defined in the following paragraphs and should be understood since the development of a multiprocessor RMX/80 exchange protocol will use many of these fields.

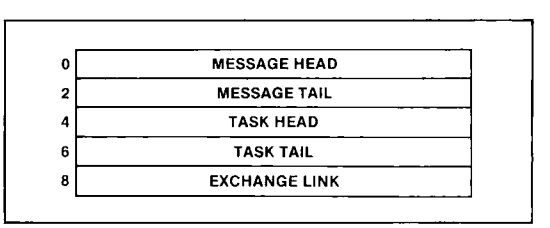

Figure 8. RMX/80™ Exchange Descriptor

MESSAGE HEAD is used to provide a pointer to the first message which has been placed into the exchange. If no messages are present at an exchange, the value of the field will be set to zero. As will be seen, this field can be tested by a task to determine if a message is already

waiting at an exchange when the task is to accept a message.

MESSAGE TAIL provides a pointer to the last message waiting at an exchange. If no messages are waiting, the pointer will be set to the address of the exchange. The multiprocessor interface which will be designed must provide an update capability for maintaining this field.

TASK HEAD is a pointer to the first task which is waiting at the exchange for a message. If no task is waiting, this field will be set to zero. Again, a simple test, when sending a message to see if a task is to be activated, is to test the field for a zero value.

TASK TAIL provides another pointer which indicates the last task which is waiting at an exchange. As with the Message Tail, the field is set to the address of the exchange when no task is waiting at the exchange.

EXCHANGE LINK contains the address of the next Exchange Descriptor in the list of all the exchange descriptors in the system. This value is established by the RMX/80 nucleus and does not require special attention in the multiprocessor modifications.

RMX/80™ Nucleus Communications

This section of the application note will deal with the generation of an RMX/80 extension which will allow the transfer of messages between tasks which reside on different processors. Three RMX/80 operations will be supported and parallel procedures developed. These will be the nucleus functions RQWAIT, RQSEND, and RQACPT.

RQWAIT is used to allow a task to receive a message from an exchange. If a message is available, its location will be transferred to the receiving task and program execution will continue. When no message is available, the task will be placed on the wait list until such time that a message has been placed into the exchange by some other task. RQWAIT provides for the capability of allowing a maximum time during which a task will wait at an exchange for a message. The multiprocessor equivalent of this function which is explained in this application note is identified as IPWAIT. It is functionally equivalent except that no provision has been made to support a maximum time interval.

RQSEND is a procedure which allows a task to send a message to an exchange. If no tasks are waiting at the exchange, the message will be queued with any other messages which may already be waiting there. If a task is found to be waiting at the exchange, it will be placed onto the ready list so that it can compete again for processor resources. The multiprocessor equivalent defined by this note is called IPSEND.

RQACPT provides a procedure which gives a task the ability to test an exchange to see if a message is present. If one is waiting, its address will be returned to the calling task. If no message is available, a value of zero is returned. The multiprocessor equivalent is called IPACPT.

The concepts used to develop each of the three extensions are discussed in the following paragraphs. Not only should this development assist the user in creating a multiprocessor executive, but it should also help in the understanding of the operation of the RMX/80 nucleus.

Multiprocessor Message Transfers

Normal message transfers take place between tasks which are resident on the same processor board and thus are able to share a common memory area for the storage of the message's data. When a multiprocessor configuration is implemented, these tasks may not necessarily have all memory areas accessible to each other. An area of memory which is common to all processor boards must be created to implement a multiprocessor system. This memory will be hereafter referenced as GLOBAL memory. All common resources which may be needed or used by a task which might have a need to perform multiprocessor operations must use the Global memory area.

The implementation of a RMX/80 extension which provides a multiprocessor communications driver can be performed using the knowledge gained from the definitions of the fields in the messages and exchanges. The development will begin with the concepts involved in executing an exchange wait.

WAITING FOR A MESSAGE

First consider the condition in which a message is waiting at the specified exchange when the task performs a request to obtain a message from an exchange. In this case, the message can be accepted directly (and the pointer fields updated accordingly) and the task can proceed normally. The sequence of events for this type of condition can be seen in the flow chart of Figure 9. A complete listing of the code can be found in Appendix A. References will be made to lines of code which are pertinent to the discussion by using a pointer in parenthesis. Thus a reference to code at line (1.14) will indicate program 1, line 14 in the appendix.

The procedure can examine the LINK field of the message to determine if more than one message is waiting at the exchange (1.36). A zero in the field indicates that no other messages are present. The value of the LINK field should be moved into the MESSAGE HEAD field of the Exchange Descriptor (1.36). The MESSAGE TAIL field should reflect the last message. If more messages are present, the field can be left unchanged; otherwise, the field should be set to the exchange address. The address of the message is returned to the calling program, allowing it to use the data contained in the message (1.40). Note how the use of a software semaphore has been used to prevent more than one processor modifying the exchange and message descriptor data at the same time (1.24; 1.39).

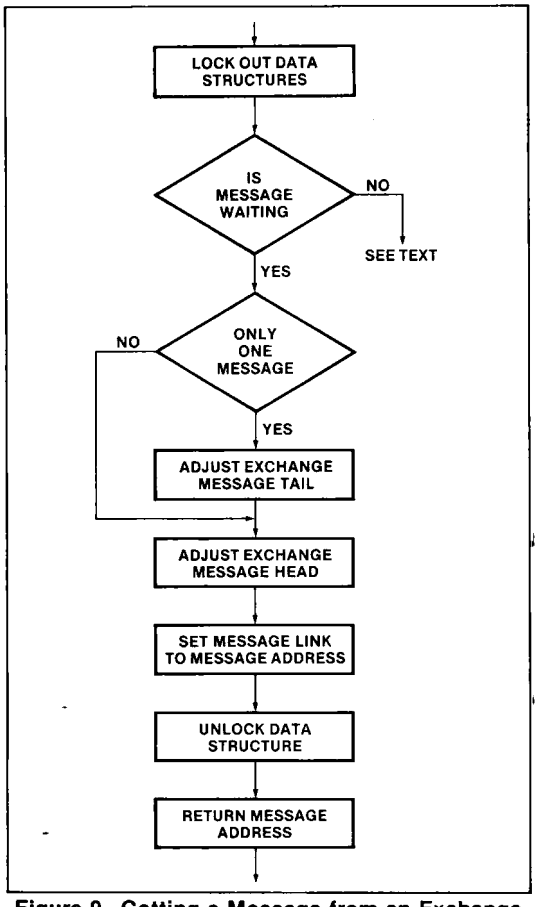

Figure 9. Getting a Message from an Exchange

The technique described above provides a direct approach to getting a message from an exchange. A more complete methodology is required when a message is not waiting at the exchange. This is the subject of the following paragraphs.

As tasks queue up at an exchange waiting for a message, provision must be made to create and maintain a link list of those tasks. Since the source code associated with RMX/80 is not normally available to the user, the effect of using the existing data structures cannot always be determined. An extension must be developed which will support the maintenance of a link list of queued tasks. Thus, each task descriptor will have a THREAD pointer added which will point to the next task waiting at the exchange. Each exchange descriptor will have two fields added, one for use as a TASK HEAD pointing to the first task waiting at the exchange, and a second field, TASK TAIL which points to the last task waiting at the exchange. These fields will be added to the existing RMX/80 descriptors for those tasks and descriptors which are associated with multiprocessor operations.

Figure 10 provides representations of the exchange and task descriptors as they would be configured for various conditions. Consider first the condition in which one task is already waiting at an exchange. The TASK TAIL and the TASK HEAD will be pointing to the task descriptor of the waiting task. Since only one task is waiting, the THREAD field of the task will contain a zero value. Appendix A contains the listing of the code required to add a task to the exchange queue. The program is the same as the one previously discussed when a message was waiting at the exchange as the calling task attempted to obtain a message. If the test indicates that no message is present (1.25), the fields must be updated

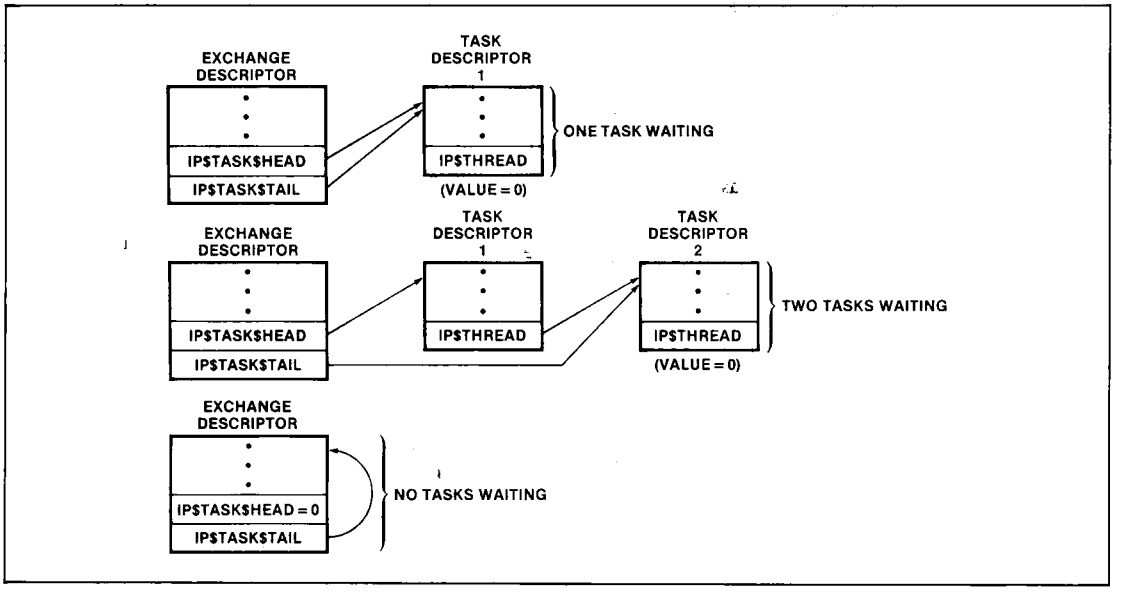

Figure 10. Task Queuing at an Exchange

as shown in Figure 8 and add the new task to the waiting list. The RMX/80 nucleus maintains the address of the task descriptor of the task currently running in a variable, RQACTV. This pointer to the current task must be placed into both the TASK TAIL of the exchange descriptor and into the THREAD of the last task queued onto the exchange (1.29). In addition, the THREAD of the current task should be set to zero to indicate that it will be the last task waiting at the exchange (1.30).

The reader, studying the program listings, might note the condition of the exchange having no tasks waiting when the task is queued up (1.28; 1.29). In this case, the TASK TAIL pointer is clearly pointing to the exchange descriptor and not to a task! The pointer normally placed into the task descriptor's THREAD field will be placed into a field of the exchange descriptor. This potential trouble area is overcome by constructing the fields of both descriptors to have offsets equivalent as shown in Figure 10. The THREAD data will now be placed into the TASK HEAD field as should be done if no tasks are already queued up.

Finally, since the task should no longer compete for system resources until a message is available to it, the program should be placed onto the delay list. However, since this list is maintained by RMX primative procedures which are unavailable to the user, the same effect can be obtained by constructing a multiprocessor user wait list and then suspending the task (1.32). At this point, the nucleus will activate another task from the ready list and the suspended task will remain in that state until it is resumed. When resumed, program execution will begin at (1.35), where a pointer from the DELAY pointer of the task descriptor will be returned to the calling program. Programs which will activate the task must ascertain that the pointer contains a reference to the message which is to be received by the task.

SENDING A MESSAGE

The conditions which may be encountered in performing a sending of a message to an exchange must now be considered. Two paths must be explored in the discussion; the operations required when a task is waiting at the exchange and the operations used when a task is not already waiting. The program which is constructed to support the required operation is called IPSEND and will have its line numbers referenced as (2.n) where n is the line number on the listing.

If the exchange to which a message is being sent does not have a task waiting (2.26), it is only necessary to modify the descriptors to indicate the presence of the message. Figure 11 shows the field pointer requirements for the various conditions which may be encountered at the exchange when no task is waiting.

An exchange having no messages already present will have the MESSAGE TAIL pointing to the exchange descriptor. A message structure of the last message at the exchange will be defined based upon the MESSAGE TAIL and the first element of the structure will be set to point to the new message (2.28; 2.29). When no message was already queued, this action will set the MESSAGE HEAD to the new message. When a message was present, the LINK field will be set to point to the new messages. Next. The LINK field of the new message will be set to zero, indicating that it is the last message in the queue (2.30). Operation of the program can then continue.

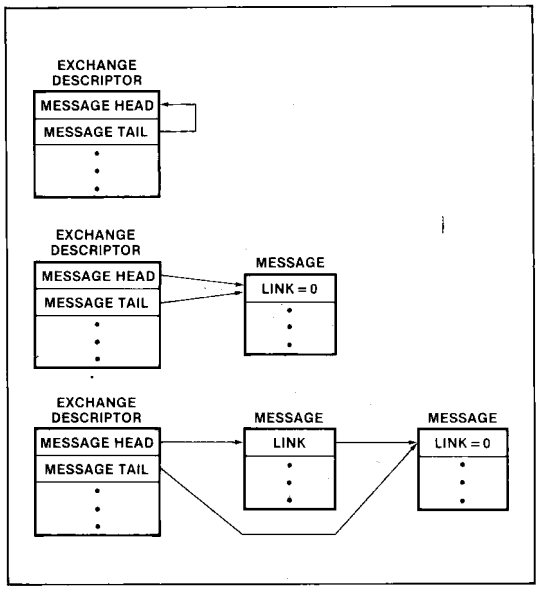

Figure 11. Placing a Message in an Exchange

The operations become more complex when one or more tasks are already waiting at an exchange for a message. In this case, the task will have been suspended as was shown in the previous discussion about waiting for a message. A technique must be established to again cause the waiting task to be placed on the ready list of the RMX/80 nucleus.

Conceptually, this can be done by generating an interrupt which will cause all processors to test if they are the one which controls the task which is waiting at the exchange to which the message has been sent. If not, no action will be taken by that board. If the task resides on the single board computer responding to the interrupt, the suspended task can be resumed and placed again on the ready list. For the purposes of this application note, interrupt level 4 will be used to provide the mechanism to cause each system processor to determine if the task to which the message is directed resides on that board.

The generation of an interrupt onto the Multibus system bus is implemented on newer iSBC boards by merely adding jumpers to wire wrap posts on the board. Figure 12 shows the schematic of the interrupt mechanism associated with interrupt level 4 on the iSBC 80/30 single board computer. An interrupt can be generated onto the Multibus system by toggling bit 7 of the I/O

port. Note that the interrupt is also routed back to the source board, so all processor boards, including the one generating the interrupt, will respond to it.

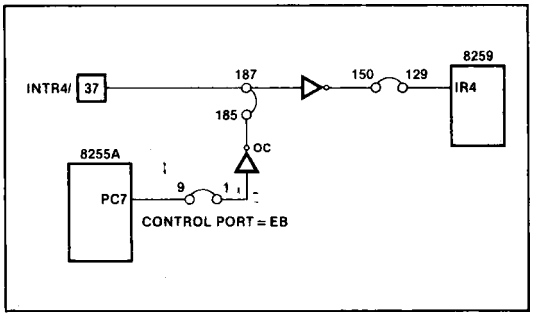

Figure 12. Interrupt Mechanism

A program has been written and is shown in Appendix A which generates the required interrupt onto the system bus. The program is identified by the name SET$INT and is referenced in this application note with the pointer scheme (3.n) where n is the line number of the code.

Of specific interest in the program listing is the use of software semaphores to provide an indication of the length of time which the interrupt line must be kept active. Semaphore 6 is used to provide mutual exclusion to the interrupt gneration mechanism (3.11) and semaphore 5 provides the synchronization mechanism. The latter semaphore is set prior to establishing an active condition on the interrupt line (3.13). When the processor having the task which is to be activated recognizes the interrupt, it will clear the semaphore level. The interrupt should be held active (low) by the processor board which generates the interrupt until the semaphore has been cleared (3.16; 3.17).

Note that in line (3.14), a processor identification number was stored in the variable PROC$ID. The technique which can be used to obtain this target processor reference is to store the processor number into the exchange descriptor STATUS field when the task is queued onto the exchange descriptor (1.27). When the message is finally sent to the exchange and the task is taken off the exchange descriptor (2.40), the identification can be stored for use by the interrupted processor. Any processor which receives an interrupt with an identification in PROC$ID which does not match its own processor identification should ignore the interrupt.

Figure 13 shows the process of dequeueing a task from the exchange descriptor. Since one or more tasks were waiting at the exchange, no messages could exist at the exchange. The first operation (2.36) involves adjusting the pointer to the first task waiting at the exchange to point to the next waiting task (if only one task was waiting, the value of zero will be stored as a pointer). If only one task is waiting (2.36), the exchange task tail is set to point to the exchange (2.37). The message link and the task delay pointers must be set to the address of the

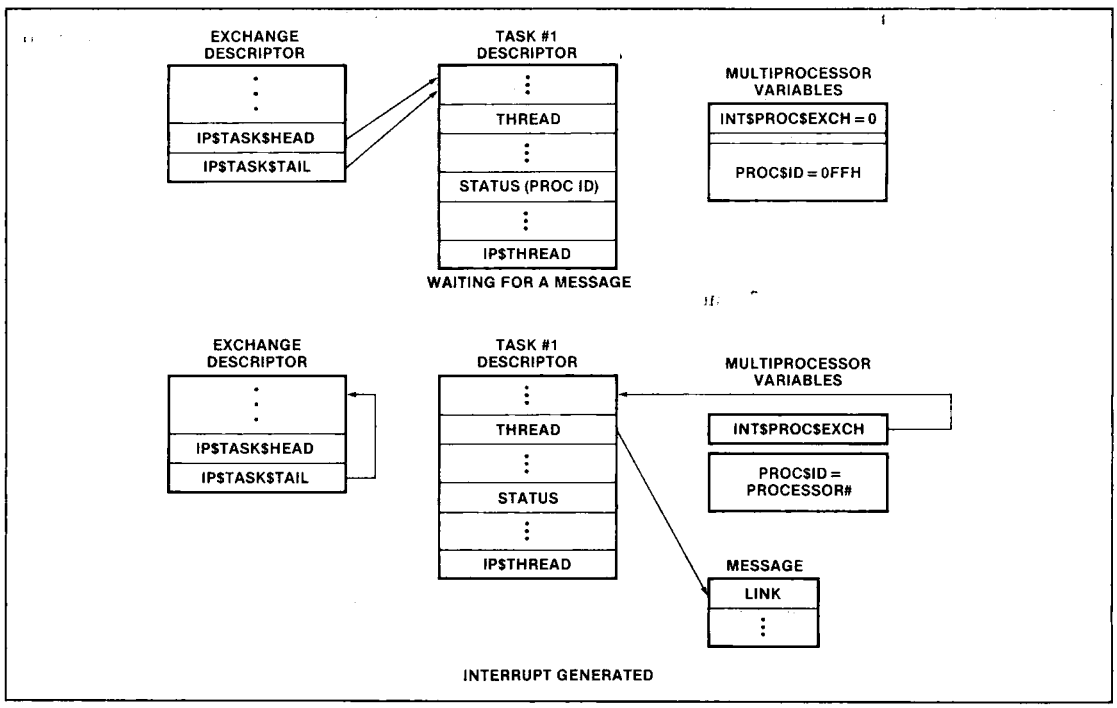

Figure 13. Dequeueing a Task

message (2.38) to support standard RMX/80 protocol. The task delay pointer is used by the receiving task to pass the address of the message when it returns to the active condition (1.33).

Before generating the interrupt which will place the waiting task onto the ready list, parameters which identify the task's processor (2.38) and internal descriptors (2.39) must be placed into a global memory area. As will be seen, this area can then be tested by each processor when an interrupt is received. An interrupt can now be generated which will cause all processors to examine the multiprocessor global memory to determine if the message is directed to a task which resides on that processor board.

RESPONDING TO INTERRUPT REQUEST

As the interrupt is generated, each processor in the system will service the request. An RMX/80 task can be written which waits at the interrupt exchange and which will service the interrupt request. However, in order to optimize the system performance, a user interrupt routine has been provided. A listing of this program, INT4, can be found in Appendix A. Code line numbers for this listing are identified by the nomenclature (5.n) where n is the line number of the code.

The program will first test the target processor identification (5.40) to determine if the task to be placed onto the ready list resides on the board. If the interrupt is found to be for another master, it will be ignored and the interrupted program will continue (5.41).

If, however, the processor is the recipient of the message, the global field containing the identification is cleared (5.43), the interrupt is acknowledged (5.44) allowing the sending master to continue the execution of its tasks, and a message is sent to the RMX/80 interrupt handler task, INTHND (5.45).

The interrupt handler continues with the INTHND procedure which is identified by the use of (6.n) where n is the line number of the code in the procedure. This task operates under the standard RMX/80 nucleus and is waiting at an interrupt exchange (6.34) for an interrupt message to be sent by the interrupt handler program, INT4. A second validity test is then performed (6.36) to verify that a multiprocessor message has actually actuated the task and that the interrupt is not to a spurious noise signal. The task pointer is saved and then cleared to allow transfer of additional messages (6.38; 6.39). The task which was waiting at the exchange is then again placed onto the ready list by performing a "resume" RMX/80 command (6.42). Thus, the transfer of a message between two tasks residing on either different or the same processors has been completed.

TESTING AN EXCHANGE

Many times it is required to test an exchange for the presence of a message without actually waiting if no message is available at the time. This is the function of the standard RMX/80 procedure RQACPT. A multiprocessing version of this function, IPACPT, was included in the extension to the RMX/80 nucleus. It is found in Appendix A and is identified by the references (4.n).

Basically, the program need only determine if a message is present at the exchange. This is done by testing the exchange message head field (4.25) for a non-zero value. If no message is present, the value will be set to zero and a return to the calling task can be performed which indicates that no message was found (4.28). If a message is found, the functions defined in IPWAIT can be executed to adjust the descriptor fields and return the message address. This is most easily accomplished by simply calling the IPWAIT procedure (4.32).

INITIALIZATION OF DESCRIPTORS

Before the multiprocessor message exchange procedures just described can function properly, various data fields must be initialized just as is done by the RMX/80 nucleus. Two programs have been written to accomplish this task. One, the GLINIT program, initializes the common global variable fields which are common to all processors. This includes such fields as the processor identification (7.14) and the target task pointer (7.9). This program also initializes the extensions which have been made to the exchange descriptors (7.10).

A second program operates on functions which are unique to each processor board in the system. For example, the I/O ports used to generate the global interrupts must be initialized to the correct state (9.43). The interrupt exchange (9.45) and interrupt handler (9.46) must be created and linked to the nucleus. Since user written interrupt service routines are used, they must also be defined to properly vector the interrupts when they occur. This is the function of lines (9.47) and (9.48).

If a board other than the iSBC 80/30 computer were used in the system, this second initialization routine would require modification to adjust it to the peculiar characteristics of the board used.

USE OF SEMAPHORES

All of the tasks which provide for multiprocessor message transfers rely upon the ability of procedures to exclude access to the RMX/80 extension which provides this capability until the message has been transferred or queued onto an exchange. This is done using semaphores in two programs identified as LOCK and UNLOCK. The same techniques which were earlier described are used and can be seen in the Appendix listings. Since the processors used in this application note were iSBC 80/30 single board computers, a bus lock command (8.17) for that board was used. The use of other boards, would require a modification of this instruction to conform to the board design. The bus lock command has no effect on memory which is accessed

using the on-board local bus. In order to provide an effective semaphore, the RAM used for a software semaphore must reside in a memory area which is external to the processor board. For this reason, the dual ported memory of an iSBC 80/30 board cannot be used and memory which is not contained on a board which uses the semaphore must be included in the system.

System Resource Sharing

Many of the advantages of a multiprocessor system can be lost when drivers and peripheral devices are duplicated for each single board computer. For example, consider a system which maintains an on-going inventory system on a mass storage device such as a disk drive. If distributed single board computers are used to support each individual operation of a product as it moves through the process, only a common disk system will allow modification of the data base by each processor. If a means can be found to also share the driver to the disk, considerable programming effort and code space can be saved.

The RMX/80 nucleus is designed to support many special I/O device handlers such as the terminal handler, and the disk file system (DFS). An intermediate level interface can be constructed which will allow any of the system processors to use common drivers on one master board. The program which will be developed in this application note for interfacing with common system drivers is called BIPI. A listing of this program can be found in Appendix B.

Several options exist when selecting a target device which is to contain all the required system resources. One possibility is to arbitrarily pick a processor board and configure the necessary drivers onto it. However, another choice seems to present many more system benefits. This is the dedication of one processor board to the operator interface and then using the iSBC 802 RMX/80 BASIC interpreter on this board. Consider, for a minute, the implications of configuring a system in this manner.

It is a fact that minimal development cost will be incurred if the programming is done in a high level language. Unfortunately, systems involving large amounts of digital input and output, or having to interface with peripheral controllers, do not lend themselves well for programming with a high level language. On the other hand, it is difficult to perform data manipulations of strings and numerical data using programs written in lower level languages. An optimum solution is then to create a system which combines the attributes of both of the language types.

An analogy can be drawn between a conventional single processor system and one which uses multiprocessors. In the system which uses a single processor, assembly language routines can be written for minimum code size or time critical functions and can be linked with PL/M or FORTRAN programs to produce the final object module. A multiprocessor system can be thought of in the same manner. Here, individual functions are represented by a complete single board computer, each of which can use some combination of the available languages. System design can emphasize the positive attributes of each language which is available.

The iSBC 802 BASIC package can be included in a multiprocessor system much the same as a common subroutine in smaller systems. Various other processor boards can use the I/O capabilities of the BASIC interpreter as needed. In particular, the DFS (Disk File System) and terminal handler can be shared as needed by each board.

Because the I/O drivers contained in the BASIC Interpreter use standard RMX/80 exchanges, they cannot be directly accessed using the multiprocessor interface which has been developed earlier in this application note. A special intermediate interface is required which will interface with the iSBC 802 software. This is the function of the program, BIPI, which will be linked together with the other tasks contained on the single board computer used for the BASIC interpreter. Certainly, the concepts which will be used to develop the interface to the common I/O services of the iSBC 802 software can easily be extended to support other packages such as the iSBC 801 FORTRAN run-time package.

Requests to have a service performed by the BIPI module will be received in the form of a message to the global exchange, BIPI$EX. A definition of services which may be required by a remote processor will provide the basis for a definition of message types which will be supported by the interface.

A fundamental request will be to gain access to the operator keyboard or display terminal. Thus two message types can be defined to read from the terminal keyboard (READ$TYPE) and to output to the display (WRITE$TYPE). To provide process tasks with the ability to examine and modify data which is stored on the disk drive media, a set of commands must also be generated which interface to DFS. Five command types will prove to be sufficient for this task. Corresponding to the RMX DFS disk message types, the program will define disk open operations (DFS$OPN) and close operations (DFS$CLS). Positioning capabilities are supported by a seek command (DFS$SK). Finally, read and write requests are supported by the read &DFS$RD) and write (DFS$WT) commands.

Examination of the program listing will indicate that disk request message types use the type numbers from 72 to 79. This provides a simple method of distinguishing a disk request from a terminal operation message. The numbers were chosen such that subtraction of 64 from the message type will yield the message type which must actually be sent to the DFS exchanges by BIPI.

In many cases, it may be desirable to suspend the BASIC program (or the FORTRAN program) while

peripheral I/O resources are being shared by another processor. An example might be when a task is reporting an alarm condition to the operator and requires some interactive communications with him. Certainly, these communications cannot be intermixed with messages from the program which was running on the terminal. Therefore, two additional commands have been provided which will suspend (SUSBAS) and resume (RESBAS) the task which might have been competing for a system resource.

Study of the program listing for BIPI will provide the reader with an insight into the use of the multiprocessing RMX/80 extensions which have been developed in this application note. Both standard RQ . . . calls and multiprocessor IP . . . calls are integrated into the task. When the concepts are clear, a real application example can be examined to see how multiprocessing can assist in the solution of an otherwise very complex design problem.

IV. APPLICATION EXAMPLE

The previous development of multiprocessing extensions to the RMX/80 nucleus allows the application example development to continue. A solution using multiprocessor techniques can be easily implemented using these concepts and the extensive line of Intel single board computer products. The results can be examined to verify that multiprocessing has provided a better solution than would have been gained if only one processor had been used.

The merits and deficiencies of multiprocessing must be examined to ascertain that the application is likely to benefit from a solution which uses more than one processor. Certainly, many functions can be better performed using a higher speed microcomputer on a single board. The decision to use multiprocessing must be made on a cost/performance basis.

The application defined at the beginning of this application note provides an excellent example of the multimaster solution. Each functional section of the application is developed into an actual board implementation which inciudes all resources required for its support.

Supervisor Implementation

The supervisory functions are likely to require extensive computational functions, report generation and operator interaction. It is also likely that, as the management gains experience with the system, the functions required and the algorithms used will be constantly modified. These operations are not generally real-time driven so extremely high system throughputs will not be required. The Intel iSBC 802 BASIC package running on a single board computer will provide a solution to these requirements. In terms of hardware design, the supervisory functions can be considered without regard to any control functions. The primary interface can be through data bases stored on a mass storage device and with the "peek" and "poke" instructions where necessary. The system configuration for the supervisory function can be seen in Figure 14.

Because the supervisor will rely extensively upon the interaction with the operator and the mass storage devices, it is natural that it have associated with it the terminal handler and the DFS system. The addition of the BIPI interface task to the BASIC interpreter package will allow the control tasks to easily interface to the inventory and control data bases as required. Figure 15 shows the primary tasks which operate on the supervisor control processor. Note that there exists two operating tasks, BASIC and BIPI, along with support functions of the DFS and terminal handler. Any program which is

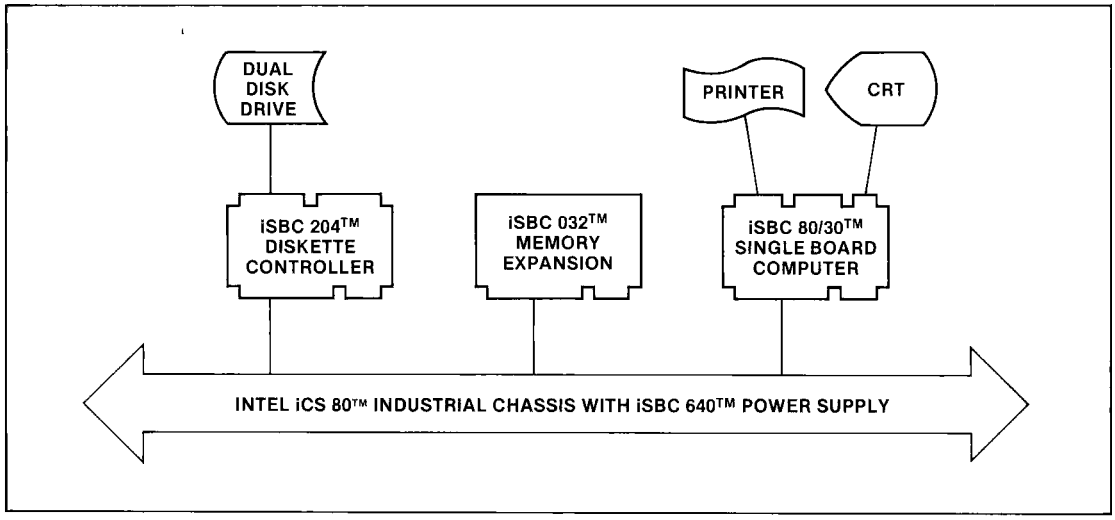

Figure 14. Supervisory Configuration

being run under BASIC will essentially become an extension of the existing interpreter task.

Programs can be written which will provide the required algorithms to implement the inventory control system and production scheduling of the system using a high level language which may easily be modified as required by the future system requirements.

The rest of the system consists of control elements. They may easily be thought of as supervisor subroutines which are called as required and which have parameters passed with the call. Unlike conventional designs, these subroutines will reside on different processor boards.

Weighing Ingredients

One functional task has been defined as weighing the ingredients according to a formula in order to provide the correct ratio of ingredients to produce the final product. As with most functional areas, a closer examination shows that this involves rather complex relationships and breaks down into many smaller tasks. Figure 16 shows the logical flow of operations which are required to weigh the ingredients into their proper ratios. From the information contained in this flow diagram, tasks and exchanges necessary to implement the function can be defined and the coding generated.

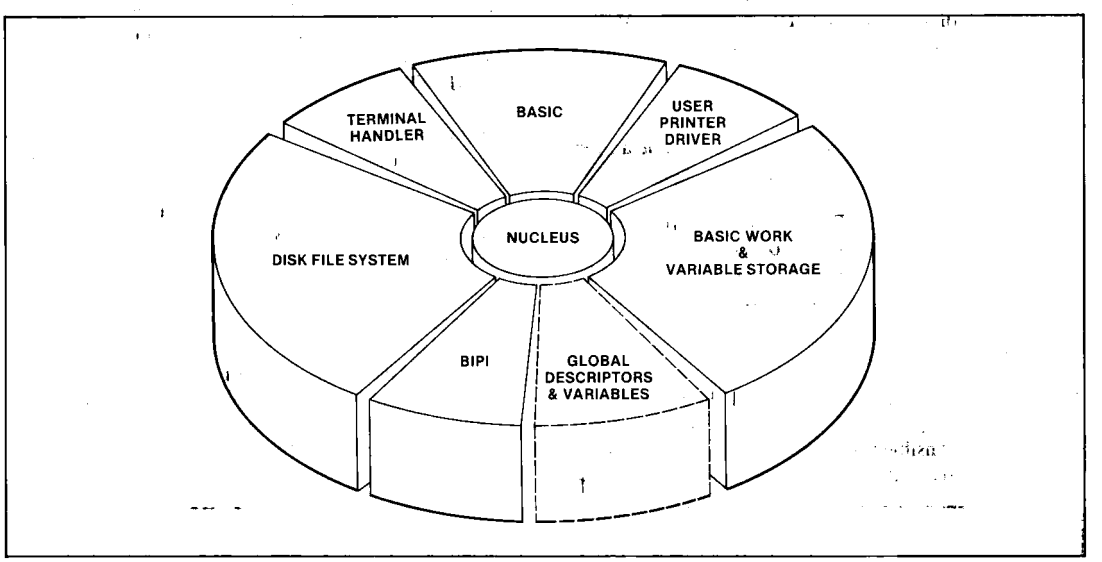

Figure 15. Supervisor Software Structure

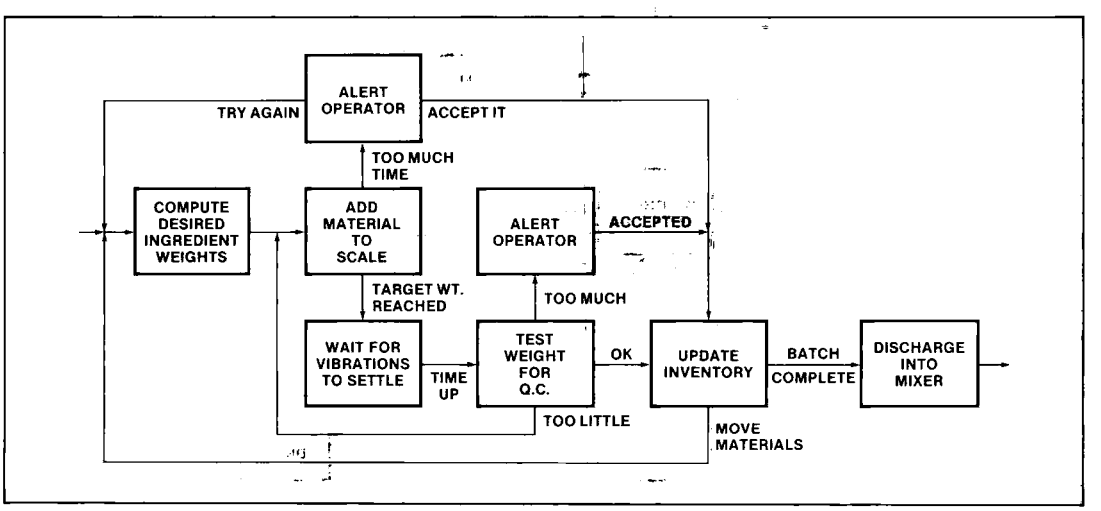

Figure 16. Weighing Logic Flow

The tasks which were defined for this application example are shown in Figure 17. The task communication paths through exchanges has also been illustrated in the figure. Basically, the weighing function consists of four on-board tasks and calls upon tasks contained on other boards as is required using the global exchanges created according to the structure defined earlier in this application note. Some time should be taken to explain the functions of each task and how they relate with tasks of other processors in the multiprocessor design solution.

The schedule task, SCHDLE, is the interface between the operator commands and the weighing functions. This task will wait for a POKE command to pass it a flag which indicates that a batch is to begin. At this point it will organize the data which will be required by the weighing task into predefined memory structures and will then signal the weighing task to begin its operations. A second function performed by the schedule task is the initialization of system variables each time a reset is performed on the processor board.

The weighing task, SCALE0, is responsible for actually producing the required amounts of each ingredient. All the functions indicated in Figure 15 are implemented by the weighing task. The task uses resources of other processors as well as on-board tasks to assist in the preparation of a batch of material.

A scale driver task, SCLWT, is included on the processor as a local support task to assist the weighing operations. It is responsible for interfacing with the analog to digital converter (ADC) and converting the data received from it into engineering units which represent the actual weight of material which is in the scale. The scale weights are passed between two tasks using an exchange (SCLWTEX) to provide mutual exclusion.

In wighing applications, many times a feeder will malfunction and not deliver material into a scale when directed to do so. If no provision is incorporated into a system design to detect this condition, an infinite time period could be required to weigh a material and, needless to say, the system throughput would be severely degraded. A slow fill detection algorithm is normally incorporated into weighing systems to provide this service. For this application, this function is generated using a slow fill task, SLOFIL. Functionally, the operation of the task is to accept a message from the weighing task that it has commenced weighing a material. SLOFIL will then begin a timed wait at an exchange for a period which corresponds with the maximum allowable time period for the material to be added into the scale. If the time elapses and a message has not been received from the weighing task indicating that cutoff has been reached, a message will be sent to the weigher indicating that a problem with the equipment exists. The operator can then be alerted to correct the malfunction.

The weighing function provides a good example of how off-board system tasks can be used to interface with those which are on-board to extend the capabilities of the system. The sharing of the terminal handler and of the DFS system has already been discussed. The weighing function calls upon these tasks, through BIPI, for assistance in updating the inventory and when it is necessary to communicate with the operator to resolve a

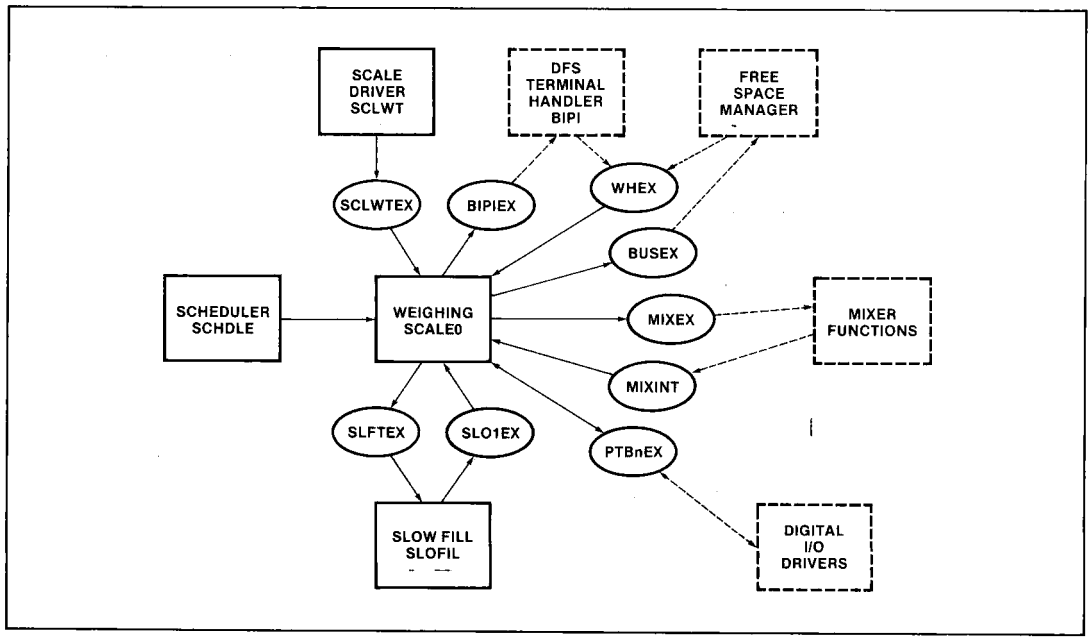

Figure 17. Weighing Task/Exchange Relationships

system problem. Other examples can also be seen. For example, the decision was made to use a common digital I/O board for all process functions. This gives the ability for access to the control and sensor devices to both the controlling tasks and to the operator for performing maintenance programs or even operating the system in a semi-automatic mode should a controller fail. All I/O data is moved through exchanges, PTBnEX, and the actual I/O is performed by a task which resides on another board.

Another example of multiprocessor communications involves message passing between the weighing tasks and the mixing tasks contained on another board. The weighing task is responsible for discharging the material into the mixer, but it certainly cannot do so unless it ascertains that the mixer is empty and available. It does this by testing the exchange, MIXINT, for the presence of a message indicating that the mixer is available. The mixer tasks, as will be seen, support this exchange by removing the message when no material should be added into the mixer and placing a message into the exchange when a mixer is ready. Further communications with the mixer are required because the mixer task must be synchronized with the discharging of material into it by the weighing function. This is provided by message transfer through global exchange, MIXEX.

Finally, because only a limited amount of RAM is available in a particular system, the free space manager was used in this application to allocate blocks of global memory to individual processor boards. This allocation is handled by sending messages to the global exchange, BUSEX, which will be received by an intermediate task on another processor and will result in a block of memory being allocated to the weighing task for message generation.

Other Application Tasks

A processor was dedicated to support each of the remaining three application processes. It would be repetitious to again detail each subtask which makes up each functional area. The same techniques which were demonstrated in defining the weighing application can be used to define the generalized functions and to break these functions into codable tasks.

Data is transferred between processors in the form of messages which occupy RAM area. The use of the RMX/80 Free Space Manager provides a global tool to allocate and reclaim this memory area. The Free Space Manager was implemented on the mixing processor because substantial amounts of ROM remained on-board after coding all the required tasks. This ability to select from many processors the location for global resources is an important tool in multiprocessor applications.

Total Application Configuration

Figure 18 shows the functional software implementation for the entire chemical application chosen for this application note. The approach has demonstrated that a relatively complex application can easily and quickly have a control solution generated using multiprocessing concepts. The ability to extend the multitasking capabilities of the RMX/80 nucleus led toward the creation of modular software.

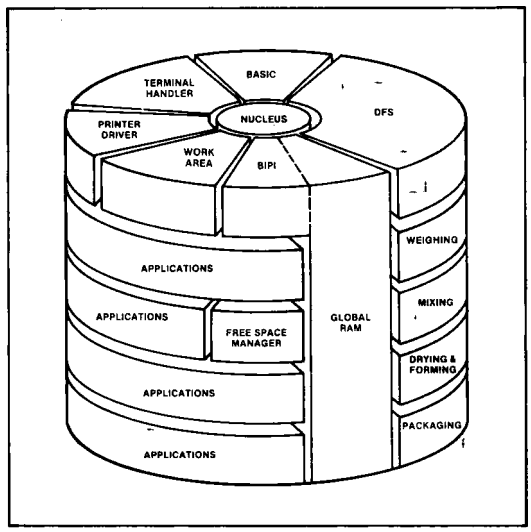

Figure 18. Software Configuration

In addition to allowing the generation of modular software, the approach taken on this example has allowed a modular approach to the hardware implementation. Figure 19 provides the complete hardware block diagram of the final implemented system. Each functional section was designed as though it were the only required element to create a solution to the control problem. Indeed, even after start-up, a functional module can be removed from the system without affecting the operation of remaining modules. The concept can be extended to include the capability of adding new processes to the total system without having to disrupt the existing production flow control.

V. CONCLUSION

This application note has demonstrated how a user can extend the concepts of multitasking into a multiprocessing/multitasking environment. The built-in multiprocessing capabilities of the various Intel single board computers have been explained and their implementations demonstrated through examples.

Resource sharing between processors has been discussed. In particular, the use of a high level language such as BASIC or FORTRAN as a supervisor has been explained and shown in an application. Since these software packages contain the resources required for disk support and terminal I/O, a program has been developed which will allow other processors to share the RMX/80 drivers.

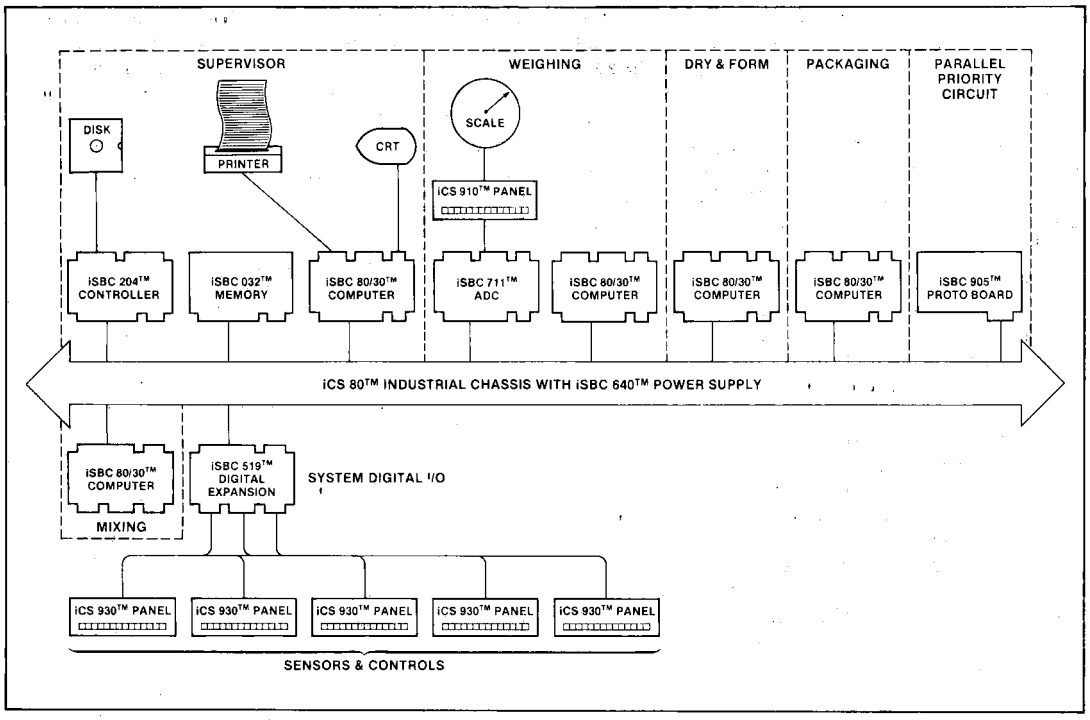

Figure 19. Hardware Block Diagram

Finally, an application has been selected, and a control solution developed using the concepts derived in the application note. The operation of the RMX/80 extensions have been tested and their operation verified.

Multimaster operations are possible because of many built-in features of the Intel products and the extensions to the RMX/80 nucleus used in this application. A summary of these features and the corresponding product is:

Multitasking Capability —

Furnished by the RMX/80 nucleus. It allows the user to share the on-board resources of a processor board between multiple tasks or functions.

Message Transfer Capabilities —

Also furnished by the RMX/80 nucleus. This function allows data to be passed between tasks to provide both an information exchange and a means of task synchronization.

Free Space Management —

The RMX/80 executive provides the ability to share memory between various tasks. Memory which is not constantly required can be used by more than one user, then returned to the memory pool.

Terminal Handler —

The RMX/80 terminal handler provides a convenient resource for supporting the communications with the operator via a CRT terminal.

Disk File System —

The RMX/80 Disk File System (DFS) provides a common resource which allows tasks to access or to store data on a diskette.

Interprocessor Communication Software —

The software developed in this application note provides an extension to the RMX/80 nucleus which allows tasks to reside on any multimaster board connected to the Multibus system bus.

Bus Arbitration Logic —

The Intel single board computers used in this application incorporate bus arbitration logic which controls the bus access to the Multibus system bus.

Parallel Priority Logic —

The application uses a parallel priority logic network which is constructed on a prototype board. In conjunction with the bus arbitration logic on each multimaster board, it controls the requests for data transfers over the Multibus system bus.

Bus Lock —

The capability of assuming exclusive control of the Multibus system bus by using the bus lock mechanism allows the implementation of software semaphores. These semaphores are used to provide mutual exclusion of certain physical resources.

BASIC Interpreter —

The use of the iSBC 802 BASIC interpreter greatly reduces the amount of time required for the generation of report programs and of interactive communications with the operator.

PL/M 80 High Level Language —

This softwasre language provides an efficient means of coding the application software where intensive bit manipulations are required. The structured language also provides a rapid development cycle.

The reader should have considerable insight into possible system level solutions to applications which he may face in his particular expertise. The increased productivity which results from applying the engineering/programming resources into application oriented efforts can easily be seen. The development costs which can be saved by using modular software and high integration level board products can certainly justify the use of these system solutions.

I would like to extend my gratitude to Steve Verleye for his valuable assistance in generating the multiprocessor communications programs used in this application note.

APPENDIX A
MULTIPROCESSOR MESSAGE TRANSFER PROGRAMS

```
/*
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        PUBLIC PROCEDURE: IPWAIT. LAST CHANGED 11/Ø7/79

   This procedure provides the inter-processor wait facility.
   If a message is available at the given exchange it is
   taken off the exchange and its address is returned to the
   calling routine. If no message is available, the processor
   ID is encoded in the status field. the task address is queued
   on the exchange list, and the task is suspended. When a
   message is sent to the exchange by another processor, the
   processor ID is used to send the task and message addresses
   into the incoming task queue for the processor and then interrupt
   it.  The service routine thus activated sets the delay field of
   the indicated task equal to the message address and RESUMEs the
   task. The task then simply RETURNs the stored message address.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                */

                $include(:f1:global.ext)
            =   /* Declaration of global inter-processor data
                   structures */
            =
   2    1   =       declare
            =           proc$id byte external,
            =           my$pid byte external,
            =           in$que address external;
   3    1   =       declare
            =           my$proc$id literally 'my$pid',
            =           int$proc$exch literally  in$que';
                $include(:f1:sema.ext)
   4    1   =   lock: procedure (sema$id) external;
   5    2   =   declare sema$id byte;
   6    2   =   end;
            =
   7    1   =   unlock: procedure (sema$id) external;
   8    2   =   declare sema$id byte;
   9    2   =   end;
                $include(suspnd.ext)
  10    1   =   RQSUSP:
            =     PROCEDURE (T$PTR) EXTERNAL;
  11    2   =       DECLARE T$PTR ADDRESS;
            =
  12    2   =     END RQSUSP;
                $include(msg.elt)
  13    1   =   DECLARE MSG$HDR LITERALLY '
            =    LINK ADDRESS,
            =    LENGTH ADDRESS,
            =    TYPE BYTE,
            =    HOME$EXCHANGE ADDRESS,
            =    RESPONSE$EXCHANGE ADDRESS';
            =
  14    1   =   DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
            =    MSG$HDR,
            =    REMAINDER(1) BYTE)';
```

```
                $include(:f1:iptask.elt)
15    1   =     DECLARE IP$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
          =      DELAY$LINK$FORWARD ADDRESS,
          =      DELAY$LINK$BACK ADDRESS,
          =      THREAD ADDRESS,
          =      DELAY ADDRESS,
          =      EXCHANGE$ADDRESS ADDRESS,
          =      SP ADDRESS,
          =      MARKER ADDRESS,
          =      PRIORITY BYTE,
          =      STATUS BYTE,
          =      NAME$PTR ADDRESS,
          =      TASK$LINK ADDRESS,
          =      IP$THREAD ADDRESS)';
                $include(:f1:ipexch.elt)
16    1   =     DECLARE IP$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE(
          =     MSG$HEAD ADDRESS
          =     MSG$TAIL ADDRESS,
          =     TASK$HEAD ADDRESS,
          =     TASK$TAIL ADDRESS,
          =     NEXT$EXCH ADDRESS,
          =     RESERVED (10) BYTE,
          =     IP$TASK$HEAD ADDRESS,
          =     IP$TASK$TAIL ADDRESS)';
                $include(common.elt)
17    1   =     DECLARE TRUE LITERALLY '0FFH';
18    1   =     DECLARE FALSE LITERALLY '00H';
19    1   =     DECLARE BOOLEAN LITERALLY 'BYTE';
20    1   =     DECLARE FOREVER LITERALLY 'WHILE 1';
                $list
21    1           declare
                      RQACTV address external;

22    1 IPWAIT: procedure (exch$adr,delay) address reentrant public;

23    2 declare
                (exch$adr,delay,temp$msg$ptr,last$task$ptr) address,
                (temp$msg based temp$msg$ptr) msg$descriptor,
                (last$task based last$task$ptr) ip$task$descriptor,
                (exch based exch$adr) ip$exchange$descriptor,
                (task based RQACTV) ip$task$descriptor;

               /* lock exchange data structure */

24    2           call lock(7);

25    2           if (temp$msg$ptr:= exch.msg$head)=0 then
26    2           do;
               /* no message at exchange, queue task up */
27    3               task.status=task.status OR shl(my$proc$id,3);
28    3               last$task$ptr=exch.ip$task$tail;
29    3               last$task.ip$thread,exch.ip$task$tail=RQACTV;
30    3               task.ip$thread=0;
31    3               call unlock(7);
32    3               call RQSUSP(RQACTV);
33    3               return task.delay;
34    3           end;

35    2           else do;
```

```
                /* There is a message here. Dequeue it and return
                   its address */

  36    3               if(exch.msg$head:= temp$msg.link) =0 then
  37    3                   exch.msg$tail=.exch;
  38    3               temp$msg.link=temp$msg$ptr;
  39    3               call unlock(7);
  40    3               return temp$msg$ptr;
  41    3           end;

  42    2       end; /* of procedure */

  43    1       end IPWAIT;

MODULE INFORMATION:
     CODE AREA SIZE     = 00F6H    246D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000AH     10D
     133 LINES READ
     0 PROGRAM ERROR(S)


            $title( 'IPSEND routine, Version 1.3')
   1         IPSEND:
                do;
             /*
            ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                PUBLIC PROCEDURE: IPSEND. LAST CHANGED 02/11/80

          This routine provides the interprocessor send facility.
          If no tasks are waiting for the message, the message is
          queued on the exchange. If a task is waiting, it is taken
          off the queue, given the address of the message, and
          queued on the interprocessor exchange.
          Following this an interrupt is generated to
          signal the responsible processor that a task is ready.


            ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                */

                $include(suspnd.ext)
   2    1  =    RQSUSP:
           =      PROCEDURE (T$PTR) EXTERNAL;
   3    2  =        DECLARE T$PTR ADDRESS;
           =
   4    2  =      END RQSUSP;
                $include(msg.elt)
   5    1  =    DECLARE MSG$HDR LITERALLY
           =     LINK ADDRESS,
           =     LENGTH ADDRESS,
           =     TYPE BYTE,
           =     HOME$EXCHANGE ADDRESS,
           =     RESPONSE$EXCHANGE ADDRESS';
           =
   6    1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
           =     MSG$HDR,
```

```
        =    REMAINDER(1) BYTE)';
             $include(:f1:iptask.elt)
 7   1  =    DECLARE IP$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
        =     DELAY$LINK$FORWARD ADDRESS,
        =     DELAY$LINK$BACK ADDRESS,
        =     THREAD ADDRESS,
        =     DELAY ADDRESS,
        =     EXCHANGE$ADDRESS ADDRESS,
        =     SP ADDRESS,
        =     MARKER ADDRESS,
        =     PRIORITY BYTE,
        =     STATUS BYTE,
        =     NAME$PTR ADDRESS,
        =     TASK$LINK ADDRESS,
        =     IP$THREAD ADDRESS)';
             include(:f1:ipexch.elt)
 8   1  =    DECLARE IP$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE(
        =    MSG$HEAD ADDRESS,
        =    MSG$TAIL ADDRESS,
        =    TASK$HEAD ADDRESS,
        =    TASK$TAIL ADDRESS,
        =    NEXT$EXCH ADDRESS,
        =    RESERVED (10) BYTE,
        =    IP$TASK$HEAD ADDRESS,
        =    IP$TASK$TAIL ADDRESS)';
             $include(common.elt)
 9   1  =    DECLARE TRUE LITERALLY '0FFH';
10   1  =    DECLARE FALSE LITERALLY '00H';
11   1  =    DECLARE BOOLEAN LITERALLY 'BYTE';
12   1  =    DECLARE FOREVER LITERALLY 'WHILE 1';
             $list
             $include(:f1:sema.ext)
13   1  =    lock: procedure (sema$id) external;
14   2  =    declare sema$id byte;
15   2  =    end;
        =
16   1  =    unlock: procedure (sema$id) external;
17   2  =    declare sema$id byte;
18   2  =    end;
             $include(:f1:global.ext)
        =    /* Declaration of global inter-processor
                data structures */
        =
19   1  =         declare
        =              proc$id byte external,
        =              my$pid byte external,
        =              in$que address external;
20   1  =         declare
        =              my$proc$id literally 'my$pid',
        =              int$proc$exch literally 'in$que';

21   1       set$int: procedure external;
22   2       end set$int;

23   1       IPSEND: procedure(exch$ptr,msg$ptr) reentrant public;

24   2           declare
                   (exch$ptr,msg$ptr,last$msg$ptr,task$ptr) address,
```

```
                    (exch based exch$ptr) ip$exchange$descriptor,
                    (msg based msg$ptr) msg$descriptor,
                    (last$msg based last$msg$ptr) msg$descriptor,
                    (task based task$ptr) ip$task$descriptor;

              /* get access to data structures */

25    2           call lock(7);

              /* check to see if any tasks are waiting */

26    2           if (task$ptr:= exch.ip$task$head)=0 then
27    2           do; /* no tasks waiting; queue message */
28    3               last$msg$ptr=exch.msg$tail;
29    3               last$msg.link.exch.msg$tail=msg$ptr;
30    3               msg.link=0;
31    3               call unlock(7);
32    3               return;
33    3           end;

34    2           else do;
              /* unqueue task and send it to proper processor */
35    3                if(exch.ip$task$head:= task.ip$thread)= 0 then
36    3                    exch.ip$task$tail=exch$ptr;
37    3                msg.link,task.delay=msg$ptr;

              /* queue task on the interprocessor exchange  */

38    3                proc$id=shr(task.status,3) and 07H;
39    3                int$proc$exch=task$ptr;
40    3                task.ip$thread=0;
41    3                call set$int;
42    3                call unlock(7);
43    3                return;
44    3           end;
45    2       end; /* of procedure */

46    1       end IPSEND;

MODULE INFORMATION:
     CODE AREA SIZE     = 01D8H    264D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000CH     12D
     130 LINES READ
     0 PROGRAM ERROR(S)


          $title('set interrupt routine')
 1         Set$int:
               do;
               /*
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

               PUBLIC PROCEDURE: SET$INT. LAST CHANGED 2/11/80.

          Set interrupt routine. Generates a level
          four interrupt on the bus. The semaphore associated
```

```
                with the proc$id field is left set until the interrupt
                is serviced. An semaphore, level 5, is set and used as
                an indicator that the interrupt has been acknowledged.

              ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                */

                $include(:f1:global.ext)
              =   /* Declaration of global inter-processor
                     data structures */
              =
    2    1    =       declare
              =           proc$id byte external,
              =           my$pid byte external,
              =           in$que address external;
    3    1    =       declare
              =           my$proc$id literally 'my$pid',
              =           int$proc$exch literally 'in$que';
                $include(:f1:sema.ext)
    4    1    =   lock: procedure (sema$id) external;
    5    2    =   declare sema$id byte;
    6    2    =   end;
              =
    7    1    =   unlock: procedure (sema$id) external;
    8    2    =   declare sema$id byte;
    9    2    =   end;

   10    1        set$int: procedure public;


   11    2            call lock(6);
   12    2            call lock(5);
   13    2            output(0EBH)=0FH;
   14    2            call lock(5);
   15    2            output(0EBH)=0EH;
   16    2            call unlock(5);
   17    2            call unlock(6);
   18    2            return;
   19    2            end;
   20    1        end set$int;

MODULE INFORMATION:
     CODE AREA SIZE     = 0022H     34D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0002H      2D
     49 LINES READ
     0 PROGRAM ERROR(S)


                $title('IPACPT routine )
    1           IPACPT:
                    do;
                /*
      ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                    PUBLIC PROCEDURE: IPACPT. LAST CHANGED 10/12/79

          This routine implements the interprocessor accept facility.
```

```
        If no message is queued at the exchange, a zero is returned.
        Else, a call is made to IPWAIT to retrieve the first message
        on the queue.

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                */

                $include(suspnd.ext)
 2    1   =     RQSUSP:
           =       PROCEDURE (T$PTR) EXTERNAL;
 3    2   =         DECLARE T$PTR ADDRESS;
           =
 4    2   =       END RQSUSP;
                $include(msg.elt)
 5    1   =     DECLARE MSG$HDR LITERALLY '
           =      LINK ADDRESS,
           =      LENGTH ADDRESS,
           =      TYPE BYTE,
           =      HOME$EXCHANGE ADDRESS,
           =      RESPONSE$EXCHANGE ADDRESS';
           =
 6    1   =     DECLARE MSG$DESCRIPTOR LITERALLY  STRUCTURE(
           =      MSG$HDR,
           =      REMAINDER(1) BYTE)';
                $include(task.elt)
 7    1   =     DECLARE TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
           =      DELAY$LINK$FORWARD ADDRESS,
           =      DELAY$LINK$BACK ADDRESS,
           =      THREAD ADDRESS,
           =      DELAY ADDRESS,
           =      EXCHANGE$ADDRESS ADDRESS,
           =      SP ADDRESS,
           =      MARKER ADDRESS,
           =      PRIORITY BYTE,
           =      STATUS BYTE,
           =      NAME$PTR ADDRESS,
           =      TASK$LINK ADDRESS)';
                $include(exch.elt)
 8    1   =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
           =      MSG$HEAD ADDRESS,
           =      MSG$TAIL ADDRESS,
           =      TASK$HEAD ADDRESS,
           =      TASK$TAIL ADDRESS,
           =      NEXT$EXCH ADDRESS)';
                $include(common.elt)
 9    1   =     DECLARE TRUE LITERALLY '0FFH';
10    1   =     DECLARE FALSE LITERALLY '00H';
11    1   =     DECLARE BOOLEAN LITERALLY 'BYTE';
12    1   =     DECLARE FOREVER LITERALLY 'WHILE 1';
                $include(:f1:sema.ext)
13    1   =     lock: procedure (sema$id) external;
14    2   =     declare sema$id byte;
15    2   =     end;
           =
16    1   =     unlock: procedure (sema$id) external;
17    2   =     declare sema$id byte;
18    2   =     end;
                $list
```

```
 19   1      IPWAIT: procedure(exch$adr,delay) address external;
 20   2          declare (exch$adr,delay) address;
 21   2      end IPWAIT;

 22   1      IPACPT: procedure(exch$ptr) address reentrant public;

 23   2          declare
                 exch$ptr address,
                 (exch based exch$ptr) exchange$descriptor;

 24   2          call lock(7);
 25   2          if exch.msg$head = 0 then
 26   2          do;
 27   3              call unlock(7);
 28   3              return 0;
 29   3          end;
 30   2          else do;
 31   3              call unlock(7);
 32   3              return IPWAIT(exch$ptr,0);
 33   3          end;
 34   2      end; /* of procedure */
 35   1      end IPACPT;
MODULE INFORMATION:
     CODE AREA SIZE     = 003BH     59D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0004H      4D
     90 LINES READ
     0 PROGRAM ERROR(S)


          $title('Level 4 interrupt handler')
           $nointvector
  1        Int$proc:
               do;

           /*
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

            PUBLIC PROCEDURE: SET$VEC$4. LAST CHANGED  2/11/80

       This routine fields all level 4 interrupts. If the global
       proc$id matches the local processor ID RQISND is called
       to awaken the int$hnd task and the interrupt semaphore is
       cleared. If the processor ID does not match, the interrupt
       is ignored.

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
           */

           $nolist

 38   1          declare
                     rql4ex (15) byte external;

 39   1      set$vec$4: procedure interrupt 4 public;

 40   2          if proc$id <> my$proc$id then
 41   2              call rqendi;
```

```
  42   2          else do;
  43   3              proc$id=0FFH;
  44   3              call unlock(5);
  45   3              call rqisnd(.rql4ex);
  46   3          end;
  47   2          return;
  48   2          end; /* of int$proc */
  49   1      end int$proc;

MODULE INFORMATION:
     CODE AREA SIZE     = 002AH     42D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000AH     10D
     110 LINES READ
     0 PROGRAM ERROR(S)


            $title('Interrupt Handler')
   1        Int$handler:
                do;
            /*
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

            PUBLIC PROCEDURE: INT$HANDLER. LAST CHANGED  2/11/80

          Interprocessor interrupt handler task. When notified by the
          Set$vec$4 routine that the In$queue contains a task-message
          pair for this processor, the Int$handler task unqueues
          the task descriptor and resumes the task where it was
          suspended.

      ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
            */

            $include(:f1:global.ext)
          =   /* Declaration of global inter-processor
                 data structures */
          =
   2   1  =       declare
          =           proc$id byte external,
          =           my$pid byte external,
          =           in$que address external;
   3   1  =       declare
          =           my$proc$id literally 'my$pid',
          =           int$proc$exch literally 'in$que';
            $include(:f1:sema.ext)
   4   1  =   lock: procedure (sema$id) external;
   5   2  =   declare sema$id byte;
   6   2  =   end;
          =
   7   1  =   unlock: procedure (sema$id) external;
   8   2  =   declare sema$id byte;
   9   2  =   end;
            $include(synch.ext)
  10   1  =   RQSEND:
          =    PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
  11   2  =       DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
          =
```

```
12    2  =      END RQSEND;
             =
13    1  =    RQWAIT:
             =      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
14    2  =        DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
             =
15    2  =      END RQWAIT;
             =
16    1  =    RQACPT:
             =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
17    2  =        DECLARE EXCHANGE$POINTER ADDRESS;
             =
18    2  =      END RQACPT;
             =
19    1  =    RQISND:
             =      PROCEDURE (IED$PTR) EXTERNAL;
20    2  =        DECLARE IED$PTR ADDRESS;
             =
21    2  =      END RQISND;
           $include(common.elt)
22    1  =    DECLARE TRUE LITERALLY '0FFH';
23    1  =    DECLARE FALSE LITERALLY '00H';
24    1  =    DECLARE BOOLEAN LITERALLY 'BYTE';
25    1  =    DECLARE FOREVER LITERALLY 'WHILE 1';
           $include(resume.ext)
26    1  =    RQRESM:
             =      PROCEDURE (T$PTR) EXTERNAL;
27    2  =        DECLARE T$PTR ADDRESS;
             =
28    2  =      END RQRESM;
           $include(:f1:iptask.elt)
29    1  =    DECLARE IP$TASK$DESCRIPTOR LITERALLY 'STRUCTURE
             =     DELAY$LINK$FORWARD ADDRESS,
             =     DELAY$LINK$BACK ADDRESS,
             =     THREAD ADDRESS,
             =     DELAY ADDRESS,
             =     EXCHANGE$ADDRESS ADDRESS,
             =     SP ADDRESS,
             =     MARKER ADDRESS,
             =     PRIORITY BYTE,
             =     STATUS BYTE,
             =     NAME$PTR ADDRESS,
             =     TASK$LINK ADDRESS,
             =     IP$THREAD ADDRESS)';

30    1           declare
                      rql4ex (15) byte external;

31    1       int$hnd: procedure public;

32    2           declare
                      (msg$ptr,task$ptr) address;

33    2           do forever;

34    3               msg$ptr=rqwait(.rql4ex,0);
35    3               call lock(7);
36    3               if int$proc$exch <> 0 then
37    3                   do;
```

```
 38    4                    task$ptr = int$proc$exch;
 39    4                    int$proc$exch = 0;
 40    4                end;
                      else
 41    3                    call unlock(7);
 42    3                    call rqresm(task$ptr);
 43    3            end; /* of do forever */
 44    2        end; /* of task */
 45    1        end int$handler;

MODULE INFORMATION:
     CODE AREA SIZE      = 003DH     61D
     VARIABLE AREA SIZE  = 0004H      4D
     MAXIMUM STACK SIZE  = 0002H      2D
     108 LINES READ
     0 PROGRAM ERROR(S)


              $title('Global Initialization. Version 1.1')
   1           global$init:
                   do;

               /*                                              *\
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

         PUBLIC PROCEDURE: GLOBAL$INIT. LAST CHANGED 02/11/80

         This is an initialization routine that is called by only
         one of the procesoors in the system. It initializes those
         structures that must be initialized before any IP routines
         are used and must then be left alone.

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
               */

               $include(:f1:ipexch.elt)
   2    1   =   DECLARE IP$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE(
            =   MSG$HEAD ADDRESS,
            =   MSG$TAIL ADDRESS,
            =   TASK$HEAD ADDRESS,
            =   TASK$TAIL ADDRESS,
            =   NEXT$EXCH ADDRESS,
            =   RESERVED (10) BYTE,
            =   IP$TASK$HEAD ADDRESS,
            =   IP$TASK$TAIL ADDRESS)';
                $include(:f1:global.ext)
            =   /* Declaration of global inter-processor
                   data structures */
            =
   3    1   =       declare
            =           proc$id byte external,
            =           my$pid byte external,
            =           in$que address external;
   4    1   =       declare
            =           my$proc$id literally 'my$pid',
            =           int$proc$exch literally 'in$que';
```

```
 5   1      declare
              ip$exch$tab address external,
              ip$exch (1) ip$exchange$descriptor at(.ip$exch$tab),
              num$ip$exch byte external;
 6   1        declare semaphore(8) byte external;

 7   1      global$init: procedure public;

 8   2          declare i byte;

            /* initialization the int$proc$exch queue */

 9   2              int$proc$exch=0;
            /* initialize user ip$exch decsriptors */
10   2          do i=0 to num$ip$exch-1;
11   3              ip$exch(i).ip$task$head=0;
12   3              ip$exch(i).ip$task$tail=.ip$exch(i);
13   3          end;

            /* initialize proc$id field */

14   2          proc$id=0FFH;

            /* initialize semaphores */

15   2          do i=0 to 7;
16   3              semaphore(i)=0;
17   3          end;

18   2          return;
19   2          end;

20   1      end global$init;

MODULE INFORMATION:
     CODE AREA SIZE     = 0075H    117D
     VARIABLE AREA SIZE = 0001H      1D
     MAXIMUM STACK SIZE = 0002H      2D
     73 LINES READ
     0 PROGRAM ERROR(S)


            $title('LOCK and UNLOCK procedures, Version 1.1 )
 1          SEMA:
              do;

            /*
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

          PUBLIC PROCEDURE: LOCK,UNLOCK. LAST CHANGED 10/12/79

          These procedures provide mutual exclusion on access
          to global data structures used in the interprocessor
          communication package. Both routines use the seventh
          semaphore for global data and the sixth semaphore for
          interrupts. Software semaphores are used.

     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
                    */
                    $include(exch.elt)
     2    1   =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
              =      MSG$HEAD ADDRESS,
              =      MSG$TAIL ADDRESS,
              =      TASK$HEAD ADDRESS,
              =      TASK$TAIL ADDRESS,
              =      NEXT$EXCH ADDRESS)';

     3   =1     rqwait: procedure(exch$ptr,delay$time) address external;
     4    2          declare (exch$ptr,delay$time) address;
     5    2     end rqwait;

     6    1       s$mask: procedure(mask) external;
     7    2           declare mask byte;
     8    2       end s$mask;

     9    1       declare timout exchange$descriptor external;
    10    1       declare semaphore(8) byte external;


                /* lock procedure called upon entry of code modifying
                   data structures */

    11    1       lock: procedure (sema$num) reentrant public;

    12    2           declare (sema$num,sema) byte;
    13    2           declare (mem$ptr) address;

    14    2           sema=1;
    15    2           do while sema=1;
    16    3               mem$ptr=rqwait(.timout,1);
    17    3               call s$mask(0C0H);
    18    3               sema=semaphore(sema$num);
    19    3               semaphore(sema$num)=0FFH;
    20    3               call s$mask(040h);
    21    3           end;
    22    2           return;
    23    2           end; /* of LOCK */

                   /* unlock called to exit region */

    24    1       unlock: procedure(sema$num) reentrant public;

    25    2           declare sema$num byte;

    26    2           semaphore(sema$num)=0;
    27    2           return;
    28    2           end; /* of unlock */

    29    1       end SEMA;

MODULE INFORMATION:
     CODE AREA SIZE     = 005FH     95D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0006H      6D
     67 LINES READ
     0 PROGRAM ERROR(S)
```

```
                $title('IP$INIT initialization routine')
  1              Ip$init:
                     do;
                 /*
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                 PUBLIC PROCEDURE: IP$INIT. LAST CHANGED 10/11/79

                 Initialization routine. Creates RQL4EX, initializes
                 the interrupts and the interrupt handlers and creates
                 the interrupt task int$hnd.

  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                 */

                     $include(:f1:global.ext)
             =       /* Declaration of global inter-processor
                        data structures */
             =
  2    1     =           declare
             =               proc$id byte external,
             =               my$pid byte external,
             =               in$que address external;
  3    1     =           declare
             =               my$proc$id literally 'my$pid',
             =               int$proc$exch literally 'in$que';
                     $include(:f1:sema.ext)
  4    1     =       lock: procedure (sema$id) external;
  5    2     =       declare sema$id byte;
  6    2     =       end;
             =
  7    1     =       unlock: procedure (sema$id) external;
  8    2     =       declare sema$id byte;
  9    2     =       end;
                     $nolist

 34    1             int$hnd: procedure external;
 35    2             end int$hnd;

 36    1             set$vec$4: procedure external;
 37    2             end set$vec$4;

 38    1                 declare
                             int$hnd$stl literally '60',
                             int$hnd$stk (int$hnd$stl) byte,
                             int$hnd$td (20) byte,
                             rql4ex (15) byte public,
                             int$hnd$pri literally '65';

 39    1                 declare int$hnd$std static$task$descriptor data(
                             'inthnd',
                             .int$hnd,
                             .int$hnd$stk,
                             int$hnd$stl,
                             int$hnd$pri,
                             0,
                             .int$hnd$td);
```

```
40   1       ip$init: procedure public;

         /* wait for someone to perform global initialization
             and reset semaphores */

41   2           call lock(7);
42   2           call unlock(7);

43   2           output(0EBH)=80H;
44   2           output(0EBH)= 0EH;
45   2           call rqcxch(.rql4ex);
46   2           call rqctsk(.int$hnd$std);
47   2           call rqsetv(.set$vec$4,4);
48   2           call rqelvl(4);
49   2           return;
50   2       end; /* of procedure */

51   1       end ip$init;

MODULE INFORMATION:
     CODE AREA SIZE      = 003DH      61D
     VARIABLE AREA SIZE  = 005FH      95D
     MAXIMUM STACK SIZE  = 0002H       2D
     133 LINES READ
     0 PROGRAM ERROR(S)
```

APPENDIX B
BIPI PROGRAM LISTING

```
  $TITLE('RMX/BASIC MULTIPROCESSOR INTERFACE, VERSION 1.1')
/*****************************************************************
*                              BIPI                              *
*     THIS IS THE INTERFACE BETWEEN THE RMX/BASIC PROCESSOR AND  *
*  THE MULTI-PROCESSOR WORLD. THE RMX/BASIC INTERPORCESSOR       *
*  INTERFACE (BIPI) INTERFACES TO THE TERMINAL HANDLER AND TO    *
*  THE DISK IO SYSTEM. THE BIPI SUPPORT THE RECEIPT OF THE       *
*  FOLLOWING MESSAGE TYPES:                                      *
*                                                                *
*      TERMINAL HANDLER REQUESTS:                                *
*           READ$TYPE     =  8       READ FROM TERMINAL         *
*           WRITE$TYPE    = 12       WRITE TO TERMINAL          *
*                                                                *
*      DISK I/O SYSTEM REQUESTS:                                 *
*           DFS$RD        = 72       READ FROM DISK             *
*           DFS$WT        = 76       WRITE TO DISK              *
*           DFS$SK        = 77       DISK SEEK OPERATION        *
*           DFS$CLS       = 78       DISK FILE CLOSE            *
*           DFS$OPN       = 79       DISK FILE OPEN             *
*                                                                *
*      BASIC TASK COMMAND REQUESTS:                              *
*           SUSBAS        = 128      SUSPEND BASIC TASK         *
*           RESBAS        = 129      RESUME BASIC TASK          *
*                                                                *
*                                                                *
*  THE TASK WILL BE ENTERED BY SENDING THE NORMAL REQUEST MES-   *
*  SAGE TO THE GLOBAL EXCHANGE BIPI$EXCH.                        *
*****************************************************************/

   1          BIPI$MODULE: DO;

            /* DEFINE THE LITERALS */

   2    1       DECLARE READ$TYPE LITERALLY '8';
   3    1       DECLARE WRITE$TYPE LITERALLY '12';
   4    1       DECLARE DFS$RD LITERALLY '72';
   5    1       DECLARE DFS$WT LITERALLY '76';
   6    1       DECLARE DFS$SK LITERALLY '77';
   7    1       DECLARE DFS$CLS LITERALLY '78';
   8    1       DECLARE DFS$OPN LITERALLY  79';
   9    1       DECLARE SUSBAS LITERALLY '128';
  10    1       DECLARE RESBAS LITERALLY '129';

            /* DECLARATION OF INTERNAL VARIABLES */

  11    1       DECLARE (MSG$PTR,RESPONSE$PTR) ADDRESS;
  12    1       DECLARE (TYPE) BYTE;

            /* DEFINE RMX PROCEDURES USED BY THE TASK */

  13    1       RQSEND:
                  PROCEDURE (EXCH$PTR,MESSAGE$PTR) EXTERNAL;
  14    2         DECLARE (EXCH$PTR,MESSAGE$PTR) ADDRESS;
  15    2         END RQSEND;
  16    1       RQWAIT:
                  PROCEDURE (EXCH$PTR,TIMEOUT) ADDRESS EXTERNAL;
  17    2         DECLARE (EXCH$PTR,TIMEOUT) ADDRESS;
  18    2         END RQWAIT;
```

```
19   1       RQSUSP:
               PROCEDURE (TASK$DESC$PTR) EXTERNAL;
20   2         DECLARE (TASK$DESC$PTR) ADDRESS;
21   2         END RQSUSP;
22   1       RQRESM:
               PROCEDURE (TASK$DESC$PTR) EXTERNAL;
23   2         DECLARE (TASK$DESC$PTR) ADDRESS;
24   2         END RQRESM;

               /* DEFINE INTER-PROCESSOR PROCEDURES USED
                  BY THE TASK */

25   1       IPSEND:
               PROCEDURE (EXCH$PTR,MESSAGE$PTR) EXTERNAL;
26   2         DECLARE (EXCH$PTR,MESSAGE$PTR) ADDRESS;
27   2         END IPSEND;
28   1       IPWAIT:
               PROCEDURE (EXCH$PTR,DUMMY) ADDRESS EXTERNAL;
29   2         DECLARE (EXCH$PTR,DUMMY) ADDRESS;
30   2         END IPWAIT;
31   1       IPINIT:
               PROCEDURE EXTERNAL;
32   2         END IPINIT;
33   1       SET$VEC$4:
               PROCEDURE EXTERNAL;
34   2         END SET$VEC$4;

               /* DEFINE THE EXCHANGES REQUIRED BY
                  THE TASK */

             $INCLUDE (:F1:EXCH.DEF)
35   1   = DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
         =           MESSAGE$HEAD          ADDRESS,
         =           MESSAGE$TAIL          ADDRESS,
         =           TASK$HEAD             ADDRESS,
         =           TASK$TAIL             ADDRESS,
         =           EXCHANGE$LINK         ADDRESS )';
36   1   = DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
         =           MESSAGE$HEAD        ADDRESS,
         =           MESSAGE$TAIL        ADDRESS,
         =           TASK$HEAD           ADDRESS,
         =           TASK$TAIL           ADDRESS,
         =           EXCHANGE$LINK       ADDRESS,
         =           LINK              ADDRESS,
         =           LENGTH            ADDRESS,
         =           TYPE              BYTE )';

37   1         DECLARE BIPI$EX EXCHANGE$DESCRIPTOR EXTERNAL;
38   1         DECLARE RQINPX EXCHANGE$DESCRIPTOR EXTERNAL;
39   1         DECLARE RQOUTX EXCHANGE$DESCRIPTOR EXTERNAL;
40   1         DECLARE RQOPNX EXCHANGE$DESCRIPTOR EXTERNAL;
41   1         DECLARE BIPI$RE EXCHANGE$DESCRIPTOR EXTERNAL;

             /* DEFINE THE BASIC MESSAGE STRUCTURE OF
                INCOMING MESSAGES */

42   1         DECLARE MSG BASED MSG$PTR STRUCTURE (
                    LINK              ADDRESS,
                    LENGTH            ADDRESS,
```

```
                    TYPE                BYTE,
                    HOME$EXCHANGE       ADDRESS.
                    RESPONSE$EXCHANGE ADDRESS  );

                /* DEFINE LOCATION OF 'BASIC' TASK DESCRIPTOR */

43   1              DECLARE BASC$TD ADDRESS EXTERNAL;

                /* BEGINNING OF BIPI PROGRAM PROCEDURE */

44   1          BIPI:
                    PROCEDURE PUBLIC;

                /* AT INITIALIZATION WAIT FOR COMMUNICATIONS
                   PROCESSOR TO COME UP */

45   2              MSG$PTR = RQWAIT(.BIPI$RE, 100);
46   2              CALL IP$INIT;

47   2              DO WHILE 1;

                /* WAIT FOR A REQUEST FROM A PROCESSOR ON BUS */

48   3                  MSG$PTR = IPWAIT (.BIPI$EX,0);

                /* SWAP RESPONSE ADDRESSES FOR LATER USE */

49   3                  RESPONSE$PTR = MSG.RESPONSE$EXCHANGE;
50   3                  MSG.RESPONSE$EXCHANGE = .BIPI$RE;

                /* SAVE MESSAGE TYPE FOR TESTING */

51   3                  TYPE = MSG.TYPE;

                /* TEST FOR A READ REQUEST FROM TERMINAL */

52   3                  IF TYPE = READ$TYPE
                        THEN CALL RQSEND (.RQINPX,MSG$PTR);

                /* TEST FOR A WRITE REQUEST TO TERMINAL */

54   3                  IF TYPE = WRITE$TYPE
                        THEN CALL RQSEND (.RQOUTX,MSG$PTR);

                /* CONVERT REQUEST TYPE TO DISK COMPATIBLE TYPE */

56   3                  IF TYPE > 71
                        THEN MSG.TYPE = MSG.TYPE - 64;

                /* HANDLE OPEN DISK FILE REQUEST */

58   3                  IF TYPE = DFS$OPN
                        THEN CALL RQSEND (.RQOPNX, MSG$PTR);

                /* SUPPORT ALL OTHER REQUEST TYPES FOR DISK I/O */

60   3                  IF ((TYPE > 71) AND (TYPE < 79)) THEN
                        CALL RQSEND (MSG.HOME$EXCHANGE,MSG$PTR);
```

```
                /* SUPPORT REQUESTS FOR 'BASIC' TASK OPERATIONS */

 62    3                IF TYPE > 127
                        THEN DO;

                /* SUSPEND TASKS IN OPERATION */

 64    4                    IF TYPE = SUSBAS
                            THEN CALL RQSUSP (BASC$TD);

                /* RESUME TASKS INTO OPERATION */

 66    4                    IF TYPE = RESBAS
                            THEN CALL RQRESM (BASC$TD);

 68    4                    END;

                /* WAIT FOR RESPONSE FOR NON-'BASIC' OPERATIONS */

 69    3                ELSE MSG$PTR = RQWAIT (.BIPI$RE, 0);

                /* SWAP RESPONSE EXCHANGES AGAIN BEFORE
                   RETURNING MESSAGE */

 70    3                MSG.RESPONSE$EXCHANGE = RESPONSE$PTR;

                /* RETURN THE MESSAGE TO THE CALLING PROGRAM */

 71    3                CALL IPSEND (RESPONSE$PTR, MSG$PTR);

                /* END OF TASK */

 72    3                END;
 73    2            END BIPI;
 74    1        END BIPI$MODULE;


MODULE INFORMATION:

     CODE AREA SIZE     = 0101H     257D
     VARIABLE AREA SIZE = 0005H       5D
     MAXIMUM STACK SIZE = 0002H       2D
     206 LINES READ
     0 PROGRAM ERROR(S)
```