intel®

APPLICATION NOTE

AP-33

October 1977

RMX/80™ Real-Time Multitasking Executive

Thomas Rolander
OEM Microcomputer
Systems Applications

9800577A

RMX/80 Real-Time Multitasking Executive

Contents

INTRODUCTION

A large number of microcomputer applications require the ability to respond to events in real time. RMX/80 provides the system software around which you can build a real-time multitasking application on Intel SBC 80 Single Board Computers. In addition, RMX/80 increases the utilization of a Single Board Computer by allowing its resources to be shared among several tasks executing concurrently. Synchronization of these multiple real-time tasks is handled by RMX/80, freeing you to concentrate your major programming efforts on your application.

This application note begins with an overview of RMX/80. Readers who are familiar with the material presented in the RMX/80 User's Guide may choose to skip to the next section, a description of how to use RMX/80 and the steps involved in using it by describing two applications.

- An interrupt driven minimal terminal handler for a CRT or Teletypewriter.
- A closed-loop analog control subsystem utilizing the Intel SBC 711 analog-to-digital board.

Each example has diagrams illustrating the relationships between its tasks and exchanges. These are useful tools in conceptualizing the activities taking place in real time. Program listings of the applications are interspersed with text describing the application.

OVERVIEW

Real-time systems provide the ability to control and respond to events occurring asynchronously in the physical world. Later in this application note, a process control application is described that monitors and controls the temperature within several chambers. The system controls the process by simply turning on and off a heat source. The system could also display the temperature on an operator's console and permit entry of new setpoint temperatures and error ranges.

A single large program could have been used to perform the functions in a sequential manner. However, this approach may not permit an operator to enter control variables at the same time the process is being monitored and controlled. In contrast, real-time systems do not operate sequentially. A number of events may all be happening at the same time. This concurrence of events is a distinguishing characteristic of real-time systems.

BASIC CONCEPTS

There are basically three concepts that the user must master to effectively use RMX/80. The first is the task, an independent program which competes for resources within the system. The second concept is the message. Messages convey data and synchronization information between tasks. The third concept is the exchange. An exchange enables one task to send a message to another. As we will see later, the interaction between tasks and exchanges enables the user to implement mutual exclusion, communication, and synchronization. Mutual exclusion is a technique that controls access to a shared resource such as an I/O device or a data structure.

Task

Under RMX/80, the user codes a separate program, known as a task, for each event. An arbitrary number of these tasks execute concurrently and are subject to synchronization as required by their functions. Tasks share resources such as data structures and can communicate between themselves.

Message

A message is a unit of communication between tasks. Together with the exchange mechanism, it conveys information between tasks and can synchronize their operations.

Exchange

RMX/80 uses message exchanges for task-to-task communication. An exchange is a pair of queues represented by a data structure at which messages are left by one task to be picked up by another. Tasks may send messages to an exchange, and may wait for messages at an exchange. A task which waits for a message may perform a timed or an untimed wait. A timed wait will terminate upon the receipt of a message or at the end of the specified period of time, even if it has not received a message. When a task does an untimed wait for a message it is guaranteed that the task will not execute again until a message is available for it. A representation of the exchange data structure is shown in Figure 1.

GENERAL CHARACTERISTICS

In addition to the basic concepts of tasks and exchanges, several other general characteristics of RMX/80 are relevant in this overview.

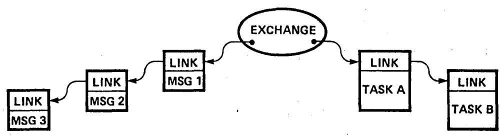

Figure 1. Exchange Data Structure

System Time Unit

RMX/80 uses a system time unit that is the period of time between "ticks" of the system clock. The standard RMX/80 system time unit is 50 milliseconds. The system time unit provides timing and user task scheduling. A task may wait at an exchange for a specified number of system time units and then continue execution. A task could be written to generate messages at specific time intervals. Tasks waiting for the messages would then be scheduled according to those time intervals.

Message Producing/Consuming Tasks

In general, tasks can be classified as message producing or message consuming tasks. The processing flow of these types of tasks are usually cyclic in nature and can be shown as follows.

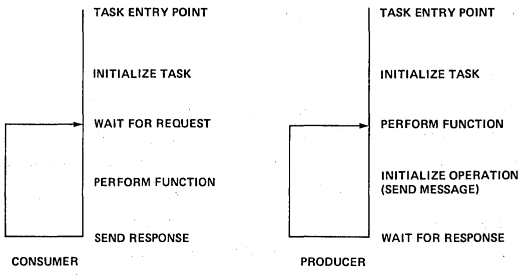

Figure 2. Message Producing/Consuming Tasks

A consumer task waits for a message to be posted at a particular exchange and takes control of the processor only when it has received a message and no other tasks of higher priority are ready to execute. The consumer task performs some action based upon the message and then simply resumes waiting until the next message is received. Usually, the consumer task acknowledges completion of its function by sending a response message to some other exchange associated with a task.

A producer task initiates its function by sending a message to another exchange and then surrenders control of the processor. The task continues to wait until it receives a response to its message.

Notice that the distinction between these types of tasks is relative since most tasks both produce and consume messages. However, the producer/consumer concept helps clarify the general structure of tasks–tasks are typically programmed loops. A producer task performs a function, sends a message, waits for a response, then loops back to begin again. A consumer task waits for a message, performs a function, sends a response, then loops back to wait again.

Interrupts

Hardware interrupts are treated as messages from peripheral devices for which a task can wait, as if the interrupt were a message from some other task. These messages arrive at particular exchanges, called interrupt exchanges, but are otherwise treated as described above. The system provides the ability to mask particular interrupts so that no messages ever arrive at a particular interrupt exchange associated with the masked interrupt. In the event that the overhead associated with turning an interrupt into a message is too high, the interrupt can be treated by the user directly via a user supplied interrupt service routine.

Task States

Tasks may exist in a number of states. A task is *running* if it actually has the processor executing instructions on its behalf. A *ready* task is one that could be running (any wait for a message or time period has been satisfied), but a higher priority task is currently running. A task is *waiting* if it cannot be ready or running because it is waiting at an exchange for a message. A *suspended* task is one that is not permitted to run or compete for system resources until it is resumed. The relationships between the task states are illustrated in Figure 3.

Priority

Each task has associated with it a priority that indicates its importance relative to other tasks in the system and relative to the interrupts of peripheral devices. RMX/80 schedules a task for execution based on the task's priority. Whenever a decision must be made on which task should be run, the highest priority ready task is chosen. Each of the eight hardware interrupt levels has a set of priorities, one of which must be assigned to the task that services the interrupt. When an interrupt occurs that task is executed if it is the highest priority ready task. At the time a higher priority task preempts a lower priority task, RMX/80 saves all the relevant information about the preempted task so it can eventually resume execution as though it were never interrupted. This process is known as a context save.

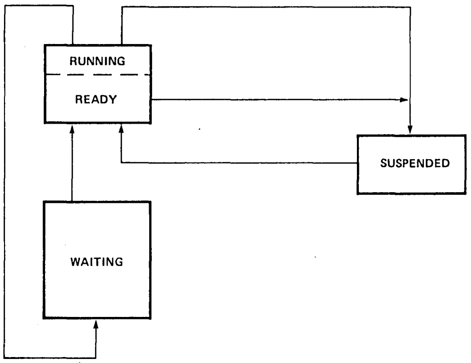

Figure 3. Task States

NUCLEUS OPERATIONS

The RMX/80 nucleus provides several operations that you can access with programmed calls. Two basic operations are covered in this section (additional operations are described in the RMX/80 User's Guide):

- RQSEND, send a message to an exchange
- RQWAIT, wait for a message or time interval

These two operations provide the capability to pass messages between tasks in a system running under RMX/80.

Message Format

The messages used by the send and wait operations to convey information between tasks are variable in length and contain the information shown in Figure 4.

0	LINK	(BASE)
2	LENGTH	
4	TYPE	
5	HOME EXCHANGE	+
7	RESPONSE EXCHANGE	+
9	REMAINDER	+

+Indicates optional

Figure 4. Message Format

Fields

1. LINK – a 2-byte field used to enter the message on a linked list at an exchange.
2. LENGTH – a 2-byte field containing the total length of the message in bytes. The minimum message length is 5 bytes (LINK, LENGTH, and TYPE).
3. TYPE – a 1-byte field indicating the type of message.
4. HOME EXCHANGE – an optional 2-byte field containing the address of an exchange to which this message should be sent when it has no further use. This field is very useful in implementing and managing a pool of messages.
5. RESPONSE EXCHANGE – an optional 2-byte field containing the address of an exchange to which a logical response to this message should be sent. This field is intended to specify the exchange at which a sending task is waiting for an acknowledgement message if one is needed.
6. REMAINDER – an optional field of arbitrary length that may contain any data portion of the message.

Sending a Message to an Exchange

The RQSEND operation enables a task to post a message at an exchange. When you send a message to an exchange, RMX/80 actually posts only the address of the message at the exchange, not the body of the message. RMX/80 avoids the overhead required to move an entire message to an exchange. Thus it is possible to queue a number of messages at the same exchange with little overhead in either execution time or memory requirements. When a task sends a message to an exchange, several functions are performed.

- The message is placed on the specified exchange.
- If there are one or more tasks waiting at the exchange, the first task is given the message and is made ready.
- If a higher priority task is thereby made ready, the sending task loses control until it once again becomes the highest priority ready task.

After a message is sent to an exchange, it must not be modified by the sending task. A task which then receives the message by waiting at the exchange where the message has been posted is free to modify the message. The format of the RQSEND operation is as follows.

RQSEND(exchange-address,message-address)

Message exchanges are defined by the user, and are normally addressed symbolically. For example, the exchange used to pass readings from an analog-to-digital (A/D) task might be named ATODEX. The reading itself could be contained in a message with the name RDNG. Thus, a typical call for a send in a PL/M program might be as follows:

```
CALL RQSEND (.ATODEX,.RDNG) ;
```

The call procedure in assembly language is as follows.

```
LXI     B,ATODEX

LXI     D,RDNG

CALL    RQSEND
```

The assembly language rules for passing parameters to RMX/80 are the same as for passing parameters to a PL/M procedure called from an assembly language module. For 2-byte parameters, the first parameter is passed in the B and C registers; the second parameter is passed in the D and E registers.

Waiting for a Message or Time Interval

The RQWAIT operation causes a task to wait for a message to arrive at an exchange. It is also possible to delay execution of a task when no message is anticipated for the task. The task simply waits for the desired time period at a message exchange where no message is ever sent. When a task waits for a message at an exchange several operations are performed.

- The task is made to wait until a message is sent to the specified exchange, or until the time limit has expired.
- When a message is available, its address is returned to the task.
- If the time limit expires before a message becomes available, a system TIME$OUT message is returned to the task.

The format of the RQWAIT operation is as follows.

RQWAIT(exchange-address,time-limit)

The time limit is entered as some number of system time units (50 milliseconds); a 1-second wait is equal to 20 time units. If zero is specified the wait is not timed, producing an indefinite wait until a message is actually sent to the exchange. Note that a specified wait of five time units may sometimes only produce an actual wait of four time units. This can occur if you enter a wait immediately before the clock "ticks." In this case the count would be decremented immediately after entering the wait. Only four full time unit periods would lapse before completion of the wait. Thus a user who wishes to ensure that at least five time units are spent in an asynchronous wait must specify six time units in the wait operation. A task which waits synchronously to the system clock, i.e., performs repetitive timed waits, does not have this problem because a new wait is executed following a tick that satisfied the previous wait. The following are typical calls for the RQWAIT operation.

PL/M

```
PTR = RQWAIT(.ATODEX,20) ;
```

The RQWAIT procedure returns an address value which is the address of a message.

Assembly Language

```
LXI     B,ATODEX

LXI     D,20

CALL    RQWAIT
```

The address of a message is returned in the HL register pair.

Send – Wait Interaction

To a large extent, the power of RMX/80 as a programming tool is derived from the interaction between send and wait. The interaction includes three multi-tasking operations.

- Communication
- Synchronization
- Mutual Exclusion

In describing these operations, a graphic notation for diagramming tasks, exchanges, and their interaction (send and wait operations) is useful. The notation is described in the next section on communication.

Communication. The most common interaction between tasks is that of communication – the transmission of data from one task to another via an exchange (Figure 5).

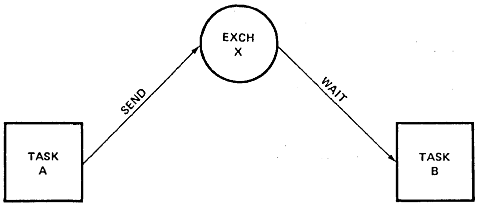

Figure 5. Communication

Rectangles designate tasks while circles represent exchanges. Arrows that are directed from tasks to exchanges indicate send operations. Wait operations are shown by arrows directed from exchanges to tasks.

Figure 5 shows an example of communication between task A and task B. Task A sends a message to exchange X and task B waits for a message at that exchange. Task A is the message producer and task B the message consumer.

Synchronization. At times there is a requirement to send a synchronizing signal from one task to another. This signal can take the form of a message that contains only header information, that is, LINK, LENGTH, and TYPE.

Let us consider the implementation of a task scheduler, used for the purposes of synchronizing another task that performs a particular function at periodic intervals. The relationship between the tasks and exchanges is shown in Figure 6.

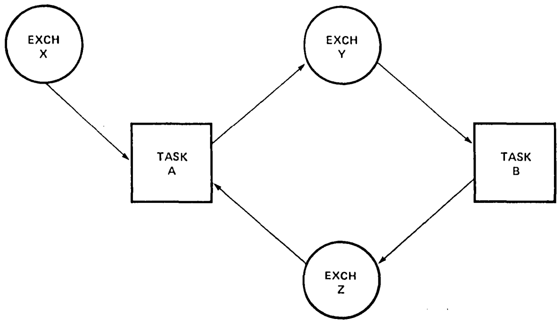

Figure 6. Synchronization

Task A, the scheduler, performs a timed wait on the X exchange. Note that the full wait period will always occur because there is no task that is sending messages to exchange X. In this manner, a specific timed wait by task A precedes the passing of a synchronization message from task A to task B via exchange Y, and then the return from task B to task A via the Z exchange.

If task B waited on X directly, rather than using task A for scheduling, it would be scheduled n system time units from when it waits instead of n units from the last time it was awakened. A comparison between the two methods is shown in Figure 7.

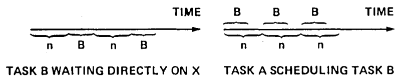

Figure 7. Scheduling Methods

Mutual Exclusion. In an environment with multitasking, resources often must be shared. Examples of shared resources include data structures and peripherals such as the Intel SBC 310 Math Module. Mutual exclusion can be used to ensure that only one task has access to a shared resource at a time. Figure 8 shows how an exchange can be used to limit access to a resource.

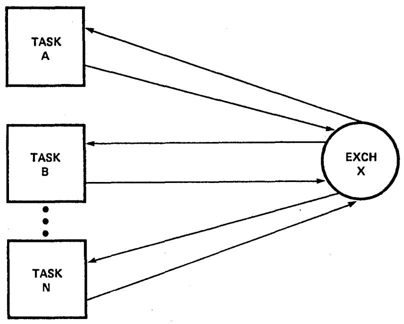

Figure 8. Mutual Exclusion

In this example, the X exchange is sent a single message at system initialization. Then, as tasks require the resource, they wait for a message from the X exchange. When the message is received, the task knows it has sole access to the resource because there is only one message associated with the exchange. After the task finishes with the resource it sends the message back to the X exchange. The next task waiting for the resource continues, knowing it has exclusive access to the resource.

EXTENSIONS

RMX/80 has several extensions which provide operations commonly used in real-time applications. The nucleus of RMX/80 requires less than two thousand bytes of memory and includes all of the basic operations. The extensions include a Free Space Manager, Terminal Handler, Disk File System, and a Debugger.

FREE SPACE MANAGER

The Free Space Manager maintains a pool of free RAM and allocates memory from that pool upon request from a task. The Free Space Manager also reclaims memory and returns it to the pool when it is no longer needed.

The Free Space Manager is especially useful in two applications. The first application arises from the need for variable length messages. If you have a task that produces messages of variable length, such as a task sending text for display on a CRT, the Free Space Manager can be used to provide a message to meet your exact size requirements.

An alternate solution is to maintain a pool of large fixed length messages. The pool can be maintained without the Free Space Manager; however, memory is wasted because of the unused space remaining in the fixed length messages.

The second application of the Free Space Manager relates specifically to effective use of memory. In a typical application, the total RAM requirement is computed by adding up the maximum RAM requirements for each task in a system as shown in Figure 9.

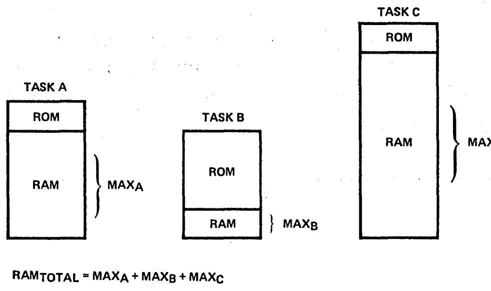

Figure 9. RAM Requirements

The efficiency of memory utilization is a function of the total RAM memory needed during typical system operation. Reducing the total amount of RAM by sharing it among the tasks often has little impact on total performance. However, significant cost advantages may be gained by reducing the total amount of memory. The memory requirements can be calculated as the minimum RAM for each task plus the pool (shared memory), as shown in Figure 10.

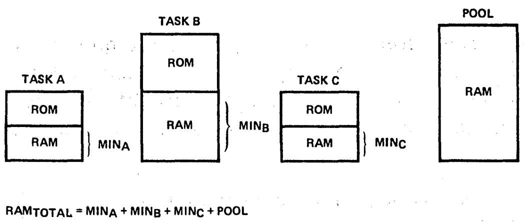

Figure 10. RAM Requirements Using a Pool

TERMINAL HANDLER

The Terminal Handler provides real-time asynchronous I/O between an operator terminal and tasks running under the RMX/80 executive. The Terminal Handler provides a line-edit capability similar to that of ISIS-II and an additional type-ahead feature. (ISIS-II is the diskette supervisor system used on the Intellec Microcomputer Development System.) Access to the Terminal Handler is provided by two exchanges where messages are sent to initiate read and write requests.

Several features of the Terminal Handler have been incorporated specifically to facilitate interaction with the debugger. Because of this interaction, the Terminal Handler is required for operation of the debugger.

DISK FILE SYSTEM

The Disk File System (DFS) provides users of RMX/80 with disk file management capabilities. This system allows user tasks to create, access, and maintain disk files in a real-time environment. This means that many I/O requests can be processed concurrently, rather than one at a time.

In addition to the file handling services, DFS provides a program loading facility that allows you to load program segments into memory from disk.

The DFS can be configured to include only those functions which you require. For example, if your disk accesses are sequential rather than random, you omit the SEEK function. This philosophy of minimizing memory requirements by including only the functions your application requires is found in virtually all aspects of RMX/80.

DEBUGGER

An environment that is continually changing in response to asynchronous physical events can present a serious debugging challenge. The Debugger aids you in debugging tasks running under the RMX/80 executive. The Debugger provides a command language that can be used to passively display information about the system, or actively modify and interact with the system.

Passive Functions

Because RMX/80 manages a fairly complex set of data structures, the Debugger has the capability of displaying them in an intelligible format. The Debugger can be used in this manner to view tasks, exchanges, messages, and other data structures maintained within the RMX/80 environment. The contents of all RAM and ROM memory locations may also be displayed by the Debugger.

Active Functions

The active Debugger functions include those of modifying memory, setting breakpoints, and monitoring stack overflow. The memory modification commands enable you to update the contents of memory and to move a series of bytes from any location to any other location.

Breakpoints can be set, allowing you to gain control when encountered by a task. Two kinds of breakpoints are supported: execution breakpoints and exchange breakpoints. An execution breakpoint can be placed at any instruction within read/write (RAM) memory. When the breakpoint is reached, the task encountering the breakpoint is stopped from further execution. The task registers may then be examined and modified before resuming execution.

Exchange breakpoints can be used to detect RQSEND and/or RQWAIT operations performed on specified exchanges. The exchange breakpoint can thus enable you to monitor the activity of any of the exchanges in your system. The task executing the appropriate RQSEND or RQWAIT to an exchange which has a breakpoint is stopped, allowing you to examine the task. This enables you to breakpoint a ROM resident task. The breakpointed task and the message involved in the operation with the exchange may then be displayed and modified before resuming execution.

The debugger can also be used to monitor stack overflow. This function is provided by the Debugger SCAN command which examines the stacks of all tasks in the system at a specified periodic interval. The fact that each task's stack is initialized with a unique value allows stack overflow to be detected. When a task stack overflows, it is removed from the system and a message is displayed.

USING RMX/80

This section of the application note describes the steps involved in using RMX/80. The process begins with the definition of the individual tasks and exchanges in your application. It continues with a discussion of the data structures that you must prepare. The task coding, compilation or assembly, linking, and locating is also described.

Finally, some comments are directed towards debugging tasks within the RMX/80 envirnment.

Before the details of using RMX/80 are discussed, some general observations are necessary to determine the suitability of RMX/80 for your application. To effectively utilize RMX/80, your application must either use interrupts or require device polling. Thus, the key element is the need to respond to external events. If your application satisfies this criteria, it is a likely candidate. However, you must then determine if RMX/80 is capable of supporting your application. This can be done by examining your interrupt response time and frequency requirements. The time required to transform an interrupt into a message that is sent to an interrupt exchange is approximately 800 microseconds for an SBC 80/20. This is the RMX/80 interrupt latency. It can be reduced to 60 microseconds by handling the interrupt directly, using the RQSETV operation to bypass the RMX/80 interrupt exchange mechanism. In this latter mode, an interrupt-driven asynchronous block transfer rate of about 10 kHz can be achieved.

TASK AND EXCHANGE DEFINITION

The initial design step for an application that runs under the RMX/80 Executive is to define your tasks, exchanges, and the interaction between them. This is perhaps best accomplished using the graphic notation introduced earlier in the section on Send – Wait Interaction. The graphic notation provides a clear picture of the relationships between the tasks and exchanges in your system. You can begin either in a top-down or bottom-up fashion. That is, you can use a top-down approach to define, at a gross level the operation of your system and then gradually break it down to the individual tasks. Or, you can start with the tasks associated with the external events in your application and then build the pieces to form the gross structure of your system.

The bottom-up approach forces you to begin with external events that drive your system. The number of these events, the amount of processing required, and the relationships between them define the tasks and exchanges in a system. For example, consider a system that samples an analog input with an A/D converter. Assume that the A/D provides an interrupt at the completion of a conversion. To use the data from the A/D converter it may also be necessary to scale it and add an offset.

With this information the portion of the task and exchange definition that relates to this function can be constructed.

Begin with the external event, the interrupt from the A/D. An interrupt priority level must be assigned to the A/D converter. This same level will be used by the task which waits on the interrupt exchange.

The relationship between the interrupt exchange and the A/D task is shown in Figure 11. If processing must be performed on raw data from the A/D, a second, lower priority, task could be used. Another task for this function will require a synchronizing signal from the ADC task to indicate that raw A/D data has been obtained and is ready for processing.

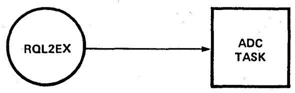

Figure 11. Interrupt Exchange and A/D Task

The interaction between the ADC task and the CONV task that processes the raw A/D data is shown in Figure 12. Two exchanges provide synchronization. The ADC task uses the TRGR exchange to signal that data is ready for processing by the CONV task. The CONV task uses the RTRGR exchange to signal the completion of its processing and thus its readiness to accept more raw data.

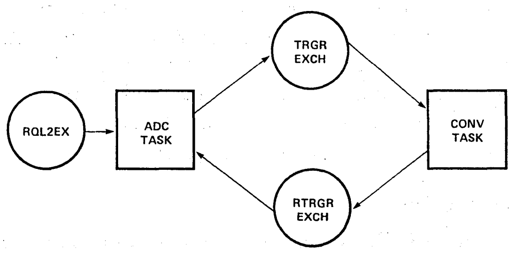

Figure 12. ADC and CONV Task Interaction

In this example two tasks and three exchanges have been defined. To develop an entire system, the tasks and exchanges associated with all of the external events in the system can be defined in the same manner. Then, proceeding bottom-up, the next step is to define the tasks and exchanges required to support the interaction between tasks running at the level of the real-time events.

After defining the entire application, you can begin actual coding of the tasks. You may choose to code in either assembly language or the PL/M 80 high-level language. Where possible, it is desirable to code in PL/M because PL/M lends itself to structured programming. Assembly language often encourages an ad hoc approach. Even if your application ultimately requires assembly language coding because of critical time and/or space parameters, initial design work in PL/M followed by translation into assembly language is recommended.

A total of 15 operations are supported by the RMX/80 nucleus. Only two of the operations, RQSEND and RQWAIT, are described in any detail in this application note. The remaining operations are described in the *RMX/80 User's Guide*. The reason for presenting only the send and wait operations is because they are sufficient for the implementation of a large number of real-time applications. These two operations provide a great deal of power and flexibility, yet their simplicity should enable those who are new to real-time programming to quickly develop applications.

PRIORITY ASSIGNMENT

The relative priority of tasks within a system running under RMX/80 determines which task is to be executed. Therefore, the assignment of a priority to each task is extremely crucial. For example, consider a compute bound task placed at a higher priority than an interrupt-driven task responsible for servicing an I/O device. This improper assignment of priorities could result in missed interrupts from the I/O device. Several steps can be followed in the assignment of task priorities.

1. Assign hardware interrupt priority levels according to the requirements of your application.
2. Specify priorities for the tasks which service the interrupts. These tasks should generally be short and serve only to perform the data transfers. A second task with a priority lower than those assigned to the hardware interrupts should be used where further processing of the data is required.
3. Priority assignment should be made for all other tasks in the system based on the relative importance and interaction among the tasks.

Unfortunately the last step in assigning task priorities is largely intuitive. In fact, you may need some empirical data from actually running your application before you settle on your final task priority assignment.

STATIC DESCRIPTORS

When a system running under RMX/80 begins execution, several tables of data are used to initialize the system. These tables usually reside in ROM. The first table is the create table (RQCRTB) that specifies the number of tasks and exchanges in the system, and the addresses of the initial task table and the initial exchange table. The initial exchange table contains the addresses of all the exchange descriptors. The initial task table contains the static task descriptors for each task, and contains the following task parameters.

1. Name
2. Initial PC – the location at which to start task execution
3. Initial SP – the location at which to start the task stack
4. Stack length
5. Priority
6. Initial Exchange (described in the *RMX/80 User's Guide*)
7. TD Address – the RAM address of the task descriptor

You must prepare all three of these tables to produce a configuration module for RMX/80. The release diskette for RMX/80 includes a set of files which contain assembly language macros that simplify the preparation of your configuration module. The relationship between these tables is shown in Figure 13.

COMPILATION/ASSEMBLY

Preparing program segments for compilation and assembly can be simplified by use of files provided on the RMX/80 diskette. As described in the last section, a set of macros is included to assist you in preparing your configuration module. Other files are provided that are useful when coding calls to RMX/80 and preparing data structures in PL/M.

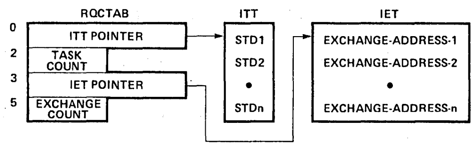

Figure 13. ROM Based Tables

By coding in a modular fashion you can separately compile and maintain tasks. This is advisable since a single large module containing all your tasks would require a lengthy recompilation to change any one of the tasks. Following the compilation and assembly of your source code modules, a library containing the object modules can be created.

LINKING

The process of linking prepares a single object module from libraries containing the RMX/80 object modules and your own application libraries or separate object modules. The order in which you specify the files to be linked is crucial to successful linking. In general, your libraries or separate object modules should be specified before the RMX/80 libraries. The link command should conclude with the unresolved library (UNRSLV.LIB) that contains miscellaneous modules for resolving PUBLICs not used in the application code. PUBLICs extend the scope of variables to allow linkage between separate program modules. Figure 14 illustrates how an application program is linked from RMX/80 and user tasks.

LOCATING

It is appropriate in this section to give some guidelines regarding the assignment of RAM and ROM address space for your Single Board Computer environment. The SBC 80 Single Board Computers have ROM based at location 0. Since the LOCATE program places all code in a contiguous block, the code must begin at location 0. Likewise, the read/write (RAM) data is also placed in a contiguous block. The base address of data should be placed at your RAM base address. Depending on the amount of code space required by your application it may be necessary to move the RAM memory base address on your SBC to a higher location. A STACKSIZE of zero should be specified because you allocate stack for each RMX/80 task in the static task descriptors.

DEBUGGING

As mentioned in the overview of the RMX/80 Debugger, the real-time environment is a complex one in which to debug your programs. Intel provides two tools that you can use for debugging. The RMX/80 Debugger and the Intel In-Circuit Emulator (ICE). It is desirable to have both of these debugging tools at your disposal.

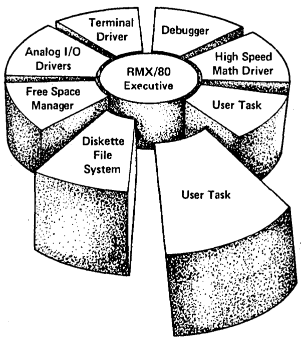

Figure 14. RMX/80 Linking

ICE enables you to use Intel Microcomputer Development System memory in place of SBC 80 memory. This allows RAM residency during your debugging as opposed to programming PROMs for each iteration. Your system may initially fail before the RMX/80 Debugger can begin operation. In this situation ICE can be used to debug your program.

APPLICATIONS

RMX/80 is suitable for a wide variety of applications. Two specific examples are presented in this application note. Each example illustrates the steps involved in using RMX/80 and provides a detailed description of the coding itself.

MINIMAL TERMINAL HANDLER

The basic functions required for a terminal handler are well defined. The handler must respond to operator input, transmit output characters, and echo characters as they are entered. This application note describes one implementation of a minimal terminal handler.

The terminal handler presented here is not the RMX/80 Terminal Handler. It does provide some common functions and uses the same exchanges and message formats. However, many features of the RMX/80 Terminal Handler have been left out. Omitted features include special hooks to run with the Debugger, an alarm exchange, control S, Q, and O operations.

As described in the chapter on using RMX/80, the process of developing an RMX/80 application begins with the definition of the tasks and exchanges. The graphic notation is used to prepare a diagram (Figure 15) showing the tasks, exchanges, and their interaction.

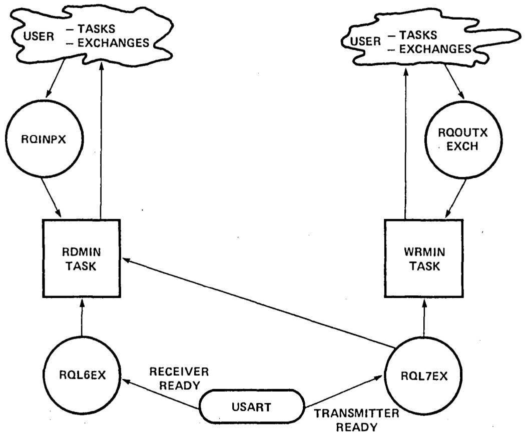

Figure 15. Minimal Terminal Handler

As shown in Figure 15, the RDMIN task waits on the RQINPX exchange for input requests. The RDMIN task also successively waits on the RQL6EX and RQL7EX exchanges. It uses the RQL6EX exchange to determine when a character has been received by the USART. The RQL7EX exchange is used to indicate when the transmitter is ready to accept another character. RDMIN uses RQL7EX for echoing input characters.

The WRMIN task waits on the RQOUTX exchange for output requests. When it receives a request, it waits on the RQL7EX to determine when characters can be sent to the USART.

The following listing* shows the RDMIN and WRMIN tasks. These tasks provide a minimal terminal handler. The program is written in PL/M. The WRMIN task is also presented in assembly language in Appendix B. The program listing is interspersed with explanatory text. The program begins with the program segment label "MINIMAL$TERMINAL$HANDLER:" and a DO statement.

```
        MINIMAL$TERMINAL$HANDLER:
        DO;
```

Several macros are declared using the reserved word LITERALLY. These macros are expanded at compile time by textual substitution.

```
        DECLARE TRUE LITERALLY '0FFH';
        DECLARE FOREVER LITERALLY 'WHILE TRUE';

        /* SPECIAL ASCII CHARACTERS */
        DECLARE
          BELL          LITERALLY '07H',
          LF            LITERALLY '0AH',
          CR            LITERALLY '0DH',
          CONTROL$R     LITERALLY '12H',
          CONTROL$X     LITERALLY '18H',
          ESC           LITERALLY '1BH',
          RUBOUT        LITERALLY '7FH';
```

Some macros are used to simplify the declaration of RMX/80 data structures. The structures declared here are for the exchange descriptor, interrupt exchange descriptor, and the messages used by the minimal terminal handler.

```
        DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
         MESSAGE$HEAD ADDRESS,
         MESSAGE$TAIL ADDRESS,
         TASK$HEAD ADDRESS,
         TASK$TAIL ADDRESS,
         EXCHANGE$LINK ADDRESS)';

        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)';

        DECLARE TH$MSG LITERALLY 'STRUCTURE(
         LINK ADDRESS,
         LENGTH ADDRESS,
         TYPE BYTE,
         HOME$EXCHANGE ADDRESS,
         RESPONSE$EXCHANGE ADDRESS,
         STATUS ADDRESS,
         BUFFER$ADDRESS ADDRESS,
         COUNT ADDRESS,
         ACTUAL ADDRESS)';
```

The following macros are specifically for the SBC 80/20. The macros require changes to run the minimal terminal handler on a different Single Board Computer. Intel 8253 timer/counter and 8251 USART chips are used.

*Full size listings in Appendixes A and C.

```
/*
  8253 PORT ADDRESSES.
*/
DECLARE A8253$MODE LITERALLY '0DFH';
DECLARE A8253$CTR2 LITERALLY '0DEH';

/*
  8253 COMMANDS.
*/
DECLARE SELECT$2 LITERALLY '10000000B';
DECLARE RL$BOTH  LITERALLY '00110000B';
DECLARE MODE$3 LITERALLY '00000110B';
DECLARE B2400 LITERALLY '001CH';

/*
  8251 PORT ADDRESSES.
*/
DECLARE USART$IN LITERALLY '0ECH',
        USART$OUT LITERALLY '0ECH',
        USART$CONTROL LITERALLY '0EDH';

/*
  8251 MODES.
*/
DECLARE STOP$1  LITERALLY '01000000B';
DECLARE CL8 LITERALLY '00001100B';
DECLARE RATE$16X LITERALLY '00000010B';

/*
  8251 COMMANDS.
*/
DECLARE USART$RESET LITERALLY '01000000B',
        RTS         LITERALLY '00100000B',
        ERROR$RESET LITERALLY '00010000B',
        RXE         LITERALLY '00000100B',
        DTR         LITERALLY '00000010B',
        TXEN        LITERALLY '00000001B';
```

RDMIN and WRMIN call three RMX/80 operations. They are RQSEND, RQWAIT, and RQELVL. RQSEND and RQWAIT allow tasks to send and receive messages from exchanges. RQELVL enables a specific interrupt level.

```
RQSEND:
  PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
    DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
  END RQSEND;

RQWAIT:
  PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
    DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
  END RQWAIT;

RQELVL:
  PROCEDURE (LEVEL) EXTERNAL;
    DECLARE LEVEL BYTE;
  END RQELVL;
```

The exchange descriptors and interrupt exchange descriptors must be PUBLIC because they are referenced by the configuration module.

```
DECLARE RQINPX EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE RQOUTX EXCHANGE$DESCRIPTOR PUBLIC;

DECLARE RQL6EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE RQL7EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
```

The following procedure initializes the 8253 and 8251 (USART). The 8253 generates the baud rate clock (2400 baud in this example). The program sends four nulls to the USART control port to ensure that the USART is ready for a command, no matter what state it was previously in. The program then sends a reset command to the USART, followed by the mode and another command.

```
INITIALIZATION:
  PROCEDURE;
    OUTPUT(A8253$MODE) = SELECT$2 OR RL$BOTH OR MODE$3;
    OUTPUT(A8253$CTR2) = LOW(B2400);

    OUTPUT(A8253$CTR2) = HIGH(B2400);
    OUTPUT(USART$CONTROL),
    OUTPUT(USART$CONTROL),
    OUTPUT(USART$CONTROL),
    OUTPUT(USART$CONTROL) = 0;
    OUTPUT(USART$CONTROL) = USART$RESET;
    OUTPUT(USART$CONTROL) = STOP$1 OR CL8 OR RATE$16X;
    OUTPUT(USART$CONTROL) = RTS OR ERROR$RESET OR
                            RXE OR DTR OR TXEN;
  END INITIALIZATION;
```

Tasks coded in PL/M take the form of parameterless PUBLIC procedures. The procedure declaration is followed by the variables used in RDMIN. MSGPTR receives the address of an input request message. The based-variable MSG accesses the data in the input request message. INTMSG is a dummy variable which simply receives the address of the interrupt message. BUF$ADDRESS points to the buffer where the input characters are to be placed. The BUF array is based at the buffer pointed to by BUF$ADDRESS.

```
RD$MIN:
  PROCEDURE PUBLIC;
    DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
    DECLARE (CHAR,PTR,I) BYTE;
    DECLARE MSG BASED MSGPTR TH$MSG;
    DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;
```

The RDMIN task echoes characters after they are read in. The ECHO$CHAR procedure performs this function. It waits for a level 7 interrupt, indicating that the transmitter is ready for another character. ECHO$CHAR then transmits the character.

```
ECHO$CHAR:
  PROCEDURE (CHAR);
    DECLARE CHAR BYTE;
    INTMSG = RQWAIT(.RQL7EX,0);
    OUTPUT(USART$OUT) = CHAR;
  END ECHO$CHAR;
```

Execution of the RDMIN task starts with the next statement, a call to the initialization procedure. This call is followed by two calls to the procedure which will enable interrupt levels 6 and 7.

```
CALL INITIALIZATION;

CALL RQELVL(6);
CALL RQELVL(7);
```

The basic structure of an RMX/80 task is that of a program with an imbedded infinite loop. This loop starts with the DO FOREVER statement. In the continuous loop, the task waits for an input request message. This wait is satisfied when some other task in the system sends an input request message to the RQINPX exchange. The based variable used to point to BUF is assigned from a field in the input request message, MSG.BUFFER$ADDRESS. An index for the BUF array, PTR, and the variable CHAR are initialized.

```
DO FOREVER;
  MSGPTR = RQWAIT(.RQINPX,0);
  BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
  PTR = 0;
  CHAR = NOT CR;
```

Task execution continues inside the next loop until a carriage return (CR) is input. An escape character (ESC) within the loop simulates a CR which enables an exit from the loop. The task simply waits on the RQL6EX exchange for a message.' This amounts to an interrupt service routine. When the wait is satisfied, the USART has received a character.

```
      DO WHILE CHAR <> CR;
        INTMSG = RQWAIT(.RQL6EX,0);
```

The next statement performs a whole series of operations. The character input from the USART is logically ANDed with 7FH to mask off the parity bit, assigned to the variable CHAR, and tested to determine if it is a RUBOUT character. If a RUBOUT is found, either a BELL is echoed to the terminal if there are not characters to delete in the buffer (PTR = 0), or the last character in the buffer is echoed and the pointer is decremented.

```
        IF (CHAR := INPUT(USART$IN) AND 7FH) = RUBOUT THEN
        DO;
          IF PTR = 0 THEN
            CALL ECHO$CHAR(BELL);
          ELSE
          DO;
            CALL ECHO$CHAR(BUF(PTR));
            PTR = PTR - 1;
          END;
        END;
```

If CHAR is not a RUBOUT, it is tested for a CONTROL$X. The function of a CONTROL$X is to delete the entire line by resetting PTR to zero. After deleting the line, the system prompts the operator with a "#" character and is ready to accept a new line.

```
        ELSE
        DO;
          IF CHAR = CONTROL$X THEN
          DO;
            CALL ECHO$CHAR('#');
            CALL ECHO$CHAR(CR);
            CALL ECHO$CHAR(LF);
            PTR = 0;
          END;
```

The next test determines if CHAR is a CONTROL$R. CONTROL$R echoes the entire line that has been entered. This function is useful for displaying a line containing a number of RUBOUTs. Such lines can be difficult to interpret because RUBOUT echoes deleted characters. Because CONTROL$R echoes only the remaining data in the buffer, it eliminates "garbage" from the display.

```
          ELSE
          DO;
            IF CHAR = CONTROL$R THEN
            DO;
              CALL ECHO$CHAR(CR);
              CALL ECHO$CHAR(LF);
              DO I = 1 TO PTR;
                CALL ECHO$CHAR(BUF(I));
              END;
            END;
```

The character is then placed in the buffer unless the end of the buffer has been reached. If the buffer is full, a BELL is sent to the terminal.

```
            ELSE
            DO;
              IF PTR < MSG.COUNT THEN
                BUF(PTR := PTR+1) = CHAR;
              ELSE
              DO;
                IF CHAR <> CR THEN
                  CHAR = BELL;
              END;
```

The last test is for an ESC character. It is echoed as a "$" and is treated as if a CR were entered.

```
              IF CHAR = ESC THEN
              DO;
                CALL ECHO$CHAR('$');
                CHAR = CR;
              END;
              CALL ECHO$CHAR(CHAR);
            END;
          END;
        END;
      END;
```

The program places a line feed (LF) at the end of the buffer when an exit is forced by a CR or an ESC. The input request message actual character count (MSG.ACTUAL) and the status (MSG. STATUS) are set before sending the message to its response exchange.

```
      IF PTR < MSG.COUNT THEN
        BUF(PTR:=PTR+1) = LF;
      MSG.ACTUAL = PTR;
      MSG.STATUS = 0;
      CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
      CALL ECHO$CHAR(LF);
    END;
  END RD$MIN;
```

The WRMIN task begins by enabling interrupt level 7. Note that no other initialization is performed before WRMIN waits for an output request message to arrive at the RQOUTX exchange. Here correct operation depends on the fact that RDMIN has a higher priority than WRMIN. Were this not the case, WRMIN could try to transmit a message before the 8253 and 8251 have been set up.

```
WR$MIN:
  PROCEDURE PUBLIC;
    DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
    DECLARE PTR BYTE;
    DECLARE MSG BASED MSGPTR TH$MSG;
    DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;

    CALL RQELVL(7);

    DO FOREVER;
      MSGPTR = RQWAIT(.RQOUTX,0);
      BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
```

The next loop transmits all of the characters specified by the output request message. Once again, the interrupt service routine is implemented by simply waiting on the RQL7EX exchange for a transmitter ready interrupt message. When this message is received, the next character in the buffer is transmitted.

```
3          DO PTR = 1 TO MSG.COUNT;
4            INTMSG = RQWAIT(.RQL7EX,0);
4            OUTPUT(USART$OUT) = BUF(PTR);
4          END;
```

The WRMIN task concludes by setting the actual count and status, and then sends the output request message to its response exchange.

```
3          MSG.ACTUAL = MSG.COUNT;
3          MSG.STATUS = 0;
3          CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
3        END;
2      END WR$MIN;
```

Using the macros provided on the RMX/80 diskette, the following static task descriptors (STD) should be placed in your configuration module.

```
STD        RDMIN,64,112,0
STD        WRMIN,64,128,0
```

The entries in the STD are interpreted as follows.

```
STD     NAME,STKLEN,PRI,EXCH
```

where:

NAME	= the symbolic name assigned to the task associated with the STD
STKLEN	= the number of bytes allocated to the task stack
PRI	= the task priority level
EXCH	= an optional field, usually 0

Priorities of 112 and 128 have been assigned to RDMIN and WRMIN because they correspond to hardware interrupt levels 6 and 7.

The following exchange addresses should be placed in your configuration module.

```
XCHADR     RQINPX
XCHADR     RQOUTX
XCHADR     RQL6EX
XCHADR     RQL7EX
```

The XCHADR macro only requires the address of the exchange descriptor.

Characters typed at the terminal are ignored unless an input request message has been received. Thus, type-ahead is not a built-in feature. However, if type-ahead is desired, it is sufficient to ensure that input requests are always queued for the RDMIN task and that the full input buffers are sent to an exchange that queues full buffers. This can easily be accomplished by sending several input requests to the RQINPX. These input requests have the address of a "full-buffer" exchange as the response exchange and the RQINPX exchange as the home exchange. Then, tasks needing terminal input wait on the "full-buffer" exchange and send the message to the home exchange when finished.

CLOSED-LOOP ANALOG CONTROL

In the next example, a closed-loop analog control subsystem using the Intel SBC 711 analog-to-digital board illustrates task scheduling and synchronization in a process control application. In general, the subsystem samples an analog input at specified intervals, converts the data to temperature in degrees centigrade, and then – based upon programmed temperature limits – controls a heating element. The algorithm used provides a 2-position controller with neutral intermediate zone (or simply "bang-bang" control). The control algorithm is shown in Figure 16.

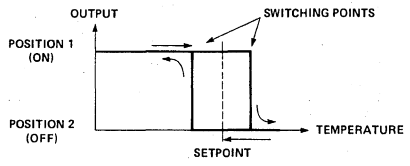

Figure 16. 2-Position Controller with Neutral Intermediate Zone

The graphic notation in Figure 17 diagrams the tasks, exchanges, and their interaction.

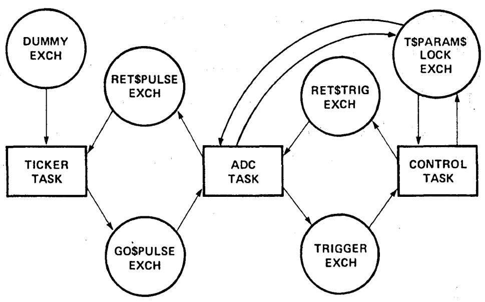

Figure 17. Analog Subsystem

This application includes three tasks and six associated exchanges. The TICKER task schedules the ADC task. TICKER has a very high priority because nothing else in the system should interfere with its scheduling activities. It is also a very short task since it repetitively executes a timed wait and then handshakes a message.

TICKER schedules the ADC task. The ADC task services the A/D converter. After obtaining data from the A/D it handshakes with the CONTROL task to signal that data is ready for processing. The ADC task is assigned a priority equivalent to the level of the hardware interrupt from the A/D. Clearly, calculations should not be performed at that priority.

Thus, CONTROL performs the processing function at a lower priority. The CONTROL task used the T$PARAM$LOCK exchange to govern access to the control parameters. This avoids problems resulting when some other task is updating the parameters at the same time the CONTROL task is using them for testing.

As in the minimal terminal handler, the following listing contains the complete analog subsystem tasks and is interspersed with explanatory text. The program begins with the program segment label "ATOD:" and a DO statement.

```
ATOD:
DO;
```

The macros and externals used in this module are brought in by means of INCLUDEs from the RMX/80 diskette.

```
$INCLUDE(:F1:COMMON.ELT)
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '00H';
DECLARE BOOLEAN LITERALLY 'BYTE';
DECLARE FOREVER LITERALLY 'WHILE 1';

$INCLUDE(:F1:EXCH.ELT)
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
 MESSAGE$HEAD ADDRESS,
 MESSAGE$TAIL ADDRESS,
 TASK$HEAD ADDRESS,
 TASK$TAIL ADDRESS,
 EXCHANGE$LINK ADDRESS)';

$INCLUDE(:F1:IED.ELT)
DECLARE INT$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)';

$INCLUDE(:F1:MSG.ELT)
DECLARE MSG$HDR LITERALLY '
 LINK ADDRESS,
 LENGTH ADDRESS,
 TYPE BYTE,
 HOME$EXCHANGE ADDRESS,
 RESPONSE$EXCHANGE ADDRESS';

DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
 MSG$HDR,
 REMAINDER(1) BYTE)';
$INCLUDE(:F1:INTRPT.EXT)
RQENDI:
  PROCEDURE EXTERNAL;

  END RQENDI;

RQELVL:
  PROCEDURE (LEVEL) EXTERNAL;
    DECLARE LEVEL BYTE;

  END RQELVL;

RQDLVL:
  PROCEDURE (LEVEL) EXTERNAL;
    DECLARE LEVEL BYTE;

  END RQDLVL;

RQSETV:
  PROCEDURE (PROC,LEVEL) EXTERNAL;
    DECLARE PROC ADDRESS;
    DECLARE LEVEL BYTE;

  END RQSETV;

$INCLUDE(:F1:SYNCH.EXT)
RQSEND:
  PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
    DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;

  END RQSEND;

RQWAIT:
  PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
    DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;

  END RQWAIT;

RQACPT:
  PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
    DECLARE EXCHANGE$POINTER ADDRESS;

  END RQACPT;

RQISND:
  PROCEDURE (IED$PTR) EXTERNAL;
    DECLARE IED$PTR ADDRESS;

  END RQISND;
```

Additional macros are declared to aid in the use of the SBC 711 analog-to-digital board.

```
/*
     SBC 711 ANALOG TO DIGITAL BOARD
*/
DECLARE ADC$BASE ADDRESS AT (0F700H);
DECLARE COMMAND$REGISTER BYTE AT (.ADC$BASE+0);
DECLARE STATUS$REGISTER BYTE AT (.ADC$BASE+0);
DECLARE FIRST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+1);
DECLARE LAST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+2);
DECLARE CLEAR$INTERRUPT$REQUEST BYTE AT (.ADC$BASE+3);
DECLARE ADC$DATA$REGISTER ADDRESS AT (.ADC$BASE+4);

DECLARE GO$BIT LITERALLY '1';
DECLARE AUTO$INCREMENT$ENABLE LITERALLY '2';
DECLARE BUSY LITERALLY '8';
DECLARE EOS$INTERRUPT$ENABLE LITERALLY '10H';
DECLARE EOC$INTERRUPT$ENABLE LITERALLY '20H';
DECLARE END$OF$SCAN LITERALLY '40H';
DECLARE END$OF$CONVERSION LITERALLY '80H';
```

The exchange descriptors and the interrupt exchange descriptors are declared.

```
DECLARE DUMMY EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE RET$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE GO$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE TRIGGER EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE RET$TRIG EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE RQL2EX INT$EXCHANGE$DESCRIPTOR;
```

The CONTROL task uses an external data structure to obtain operating parameters. This data structure (BOX$PARAMS) has an exchange associated with it (T$PARAM$LOCK) that is used to provide mutual exclusion, ensuring that only one task accesses the data structure at a time.

```
DECLARE T$PARAM$LOCK EXCHANGE$DESCRIPTOR EXTERNAL;

DECLARE BOX$PARAMS(5) STRUCTURE(
        CHANNEL BYTE,
        SET$POINT ADDRESS,
        ERROR ADDRESS,
        OFFSET ADDRESS,
        SAMPLES ADDRESS,
        COUNT ADDRESS,
        ACCUM ADDRESS,
        READING ADDRESS ) EXTERNAL;
```

TICKER, the scheduler task, has an initialization sequence in which it sets up two messages and sends them to the RET$PULSE exchange. Then it enters an infinite loop where it waits on the DUMMY exchange for 250 milliseconds. After the timed wait is complete, TICKER passes a message from the RET$PULSE exchange to the GO$-PULSE exchange. In effect this is a handshake, checking to see that the ADC task has completed its last operation and then signaling it to perform another.

```
TICKER$TASK:
  PROCEDURE PUBLIC;
    DECLARE MSG ADDRESS;
    DECLARE PULSE$MSG(2) STRUCTURE (
            MSG$HDR );

    PULSE$MSG(0).LENGTH,
    PULSE$MSG(1).LENGTH = SIZE(PULSE$MSG(0));
    PULSE$MSG(0).TYPE,
    PULSE$MSG(1).TYPE = 65;
    CALL RQSEND(.RET$PULSE,.PULSE$MSG(0));
    CALL RQSEND(.RET$PULSE,.PULSE$MSG(1));

    DO FOREVER;
      MSG = RQWAIT(.DUMMY,5);
      MSG = RQWAIT(.RET$PULSE,0);
      CALL RQSEND(.GO$PULSE,MSG);
    END;

  END TICKER$TASK;
```

Scheduled by TICKER, the ADC task performs the A/D sampling. It begins by setting up TRIGGER$-MSG and enabling the level 2 interrupt from the A/D. Inside the ADC task continuous loop, messages are passed from the GO$PULSE exchange to the RET$PULSE exchange. Then it waits for access to the BOX$PARAMS data structure. When the ADC task has access, it loops through the A/D channels, accumulating readings in BOX$PARAMS. After all the A/D channels are sampled and the BOX$PARAMS readings updated, the LOCK$MSG is returned to the T$PARAM$LOCK exchange. The ADC task concludes the continuous loop by handshaking a message with the CONTROL task.

```
ADC$TASK:
  PROCEDURE PUBLIC;
    DECLARE TRIGGER$MSG STRUCTURE (
            MSG$HDR );
    DECLARE (T$MSG,MSG,LOCK$MSG) ADDRESS;
    DECLARE I BYTE;
    DECLARE GAIN LITERALLY '00';
    DECLARE N$CHNLS LITERALLY '5';

    TRIGGER$MSG.LENGTH = SIZE(TRIGGER$MSG);
    TRIGGER$MSG.TYPE = 65;
    CALL RQSEND(.RET$TRIG,.TRIGGER$MSG);
    CALL RQELVL(2);

    DO FOREVER;
      MSG = RQWAIT(.GO$PULSE,0);
      CALL RQSEND(.RET$PULSE,MSG);
      LOCK$MSG = RQWAIT(.T$PARAM$LOCK,0);
      DO I = 0 TO N$CHNLS-1;
        FIRST$CHANNEL$REGISTER = BOX$PARAMS(I).CHANNEL
                                 + ROL(GAIN,6);
        COMMAND$REGISTER = GO$BIT
                           OR EOC$INTERRUPT$ENABLE;
        MSG = RQWAIT(.RQL2EX,0);
        COMMAND$REGISTER = 0;
        BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).ACCUM
                              + ADC$DATA$REGISTER;
      END;
      CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
      T$MSG = RQWAIT(.RET$TRIG,0);
      CALL RQSEND(.TRIGGER,T$MSG);
    END;

  END ADC$TASK;
```

The CONTROL task waits for a message from the ADC task signaling that A/D readings have been taken and are ready for further processing. It completes the handshake by sending the message to the RET$TRIG exchange. Then, as in the ADC task, accesses the BOX$PARAMS data structure.

Inside the next loop, the readings are averaged, scaled, offset, and tested. Appropriate action is taken to turn the heating elements on or off. The loop concludes by returning the message to the T$PARAM$LOCK exchange.

```
CONTROL$TASK:
  PROCEDURE PUBLIC;
    DECLARE (LOCK$MSG,T,MSG) ADDRESS;
    DECLARE I BYTE;
    DECLARE NCHNLS LITERALLY '5';
    DECLARE TURN$LAMP$ON
      LITERALLY 'OUTPUT(0E7H)=SHL(I,1)';
    DECLARE TURN$LAMP$OFF
      LITERALLY 'OUTPUT(0E7H)=SHL(I,1)+1';
    DECLARE SETUP$8255 LITERALLY 'OUTPUT(0E7H)=80H;
                                  OUTPUT(0E6H)=0FFH';

    SETUP$8255;

    DO FOREVER;
      MSG = RQWAIT(.TRIGGER,0);
      CALL RQSEND(.RET$TRIG,MSG);
      LOCK$MSG= RQWAIT(.T$PARAM$LOCK,0);
      DO I = 0 TO NCHNLS-1;
        BOX$PARAMS(I).COUNT = BOX$PARAMS(I).COUNT + 1;
        IF BOX$PARAMS(I).COUNT
          = BOX$PARAMS(I).SAMPLES THEN
        DO;
          T,
          BOX$PARAMS(I).READING
            = (BOX$PARAMS(I).ACCUM
              /BOX$PARAMS(I).SAMPLES) / 38
              + BOX$PARAMS(I).OFFSET;
          IF T <= BOX$PARAMS(I).SET$POINT
                  - BOX$PARAMS(I).ERROR THEN
            TURN$LAMP$ON;
          ELSE
            IF T >= BOX$PARAMS(I).SET$POINT
                    + BOX$PARAMS(I).ERROR THEN
              TURN$LAMP$OFF;
          BOX$PARAMS(I).ACCUM,
          BOX$PARAMS(I).COUNT = 0;
        END;
      END;
      CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
    END;

  END CONTROL$TASK;

END ATOD;
```

SUMMARY/CONCLUSIONS

The purpose of this application note is to introduce you to the Intel RMX/80, Real-Time Multitasking Executive. The general framework of RMX/80 was discussed, including the nucleus and extensions.

This application note described the steps involved in using RMX/80. Key emphasis has been placed on the need to fully define the tasks and exchanges in your application using graphic notation.

Applications have been presented to demonstrate task communication, synchronization, and mutual exclusion in a minimal terminal handler and an analog subsystem. The tasks responded to real-time asynchronous events such as USART and A/D interrupts.

RMX/80 represents a significant step in the sophistication of microcomputer software. Its ease of use, flexibility, and power should enable you to quickly implement real-time software for your applications.

APPENDIX A

MINITH PL/M LISTING

```
MINIMAL$TERMINAL$HANDLER:

DO;

DECLARE TRUE LITERALLY '0FFH';
DECLARE FOREVER LITERALLY 'WHILE TRUE';

/* SPECIAL ASCII CHARACTERS */
DECLARE
  BELL          LITERALLY '07H',
  LF            LITERALLY '0AH',
  CR            LITERALLY '0DH',
  CONTROL$R     LITERALLY '12H',
  CONTROL$X     LITERALLY '18H',
  ESC           LITERALLY '1BH',
  RUBOUT        LITERALLY '7FH';

DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
 MESSAGE$HEAD ADDRESS,
 MESSAGE$TAIL ADDRESS,
 TASK$HEAD ADDRESS,
 TASK$TAIL ADDRESS,
 EXCHANGE$LINK ADDRESS)';

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)';

DECLARE TH$MSG LITERALLY 'STRUCTURE(
 LINK ADDRESS,
 LENGTH ADDRESS,
 TYPE BYTE,
 HOME$EXCHANGE ADDRESS,
 RESPONSE$EXCHANGE ADDRESS,
 STATUS ADDRESS,
 BUFFER$ADDRESS ADDRESS,
 COUNT ADDRESS,
 ACTUAL ADDRESS)';

/*
  8253 PORT ADDRESSES.
*/
DECLARE A8253$MODE LITERALLY '0DFH';
DECLARE A8253$CTR2 LITERALLY '0DEH';
```

```
                /*
                  8253 COMMANDS.
                */
10    1         DECLARE SELECT$2 LITERALLY '1ØØØØØØØB';
11    1         DECLARE RL$BOTH  LITERALLY 'ØØ11ØØØØB';
12    1         DECLARE MODE$3 LITERALLY 'ØØØØØ11ØB';
13    1         DECLARE B24ØØ LITERALLY 'ØØ1CH';

                /*
                  8251 PORT ADDRESSES.
                */
14    1         DECLARE USART$IN LITERALLY 'ØECH',
                        USART$OUT LITERALLY 'ØECH',
                        USART$CONTROL LITERALLY 'ØEDH';

                /*
                  8251 MODES.
                */
15    1         DECLARE STOP$1  LITERALLY 'Ø1ØØØØØØB';
16    1         DECLARE CL8 LITERALLY 'ØØØØ11ØØB';
17    1         DECLARE RATE$16X LITERALLY 'ØØØØØØ1ØB';

                /*
                  8251 COMMANDS.
                */
18    1         DECLARE USART$RESET LITERALLY 'Ø1ØØØØØØB',
                        RTS         LITERALLY 'ØØ1ØØØØØB',
                        ERROR$RESET LITERALLY 'ØØØ1ØØØØB',
                        RXE         LITERALLY 'ØØØØØ1ØØB',
                        DTR         LITERALLY 'ØØØØØØ1ØB',
                        TXEN        LITERALLY 'ØØØØØØØ1B';

19    1         RQSEND:
                  PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
2Ø    2             DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
21    2           END RQSEND;

22    1         RQWAIT:
                  PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
23    2             DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
24    2           END RQWAIT;

25    1         RQELVL:
                  PROCEDURE (LEVEL) EXTERNAL;
26    2             DECLARE LEVEL BYTE;
27    2           END RQELVL;

28    1         DECLARE RQINPX EXCHANGE$DESCRIPTOR PUBLIC;
29    1         DECLARE RQOUTX EXCHANGE$DESCRIPTOR PUBLIC;

3Ø    1         DECLARE RQL6EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
31    1         DECLARE RQL7EX INT$EXCHANGE$DESCRIPTOR PUBLIC;

32    1         INITIALIZATION:
                  PROCEDURE;
33    2             OUTPUT(A8253$MODE) = SELECT$2 OR RL$BOTH OR MODE$3;
34    2             OUTPUT(A8253$CTR2) = LOW(B24ØØ);
```

```
    OUTPUT(A8253$CTR2) = HIGH(B2400);
    OUTPUT(USART$CONTROL),
    OUTPUT(USART$CONTROL),
    OUTPUT(USART$CONTROL),
    OUTPUT(USART$CONTROL) = 0;
    OUTPUT(USART$CONTROL) = USART$RESET;
    OUTPUT(USART$CONTROL) = STOP$1 OR CL8 OR RATE$16X;
    OUTPUT(USART$CONTROL) = RTS OR ERROR$RESET OR
                            RXE OR DTR OR TXEN;
  END INITIALIZATION;

RD$MIN:
  PROCEDURE PUBLIC;
    DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
    DECLARE (CHAR,PTR,I) BYTE;
    DECLARE MSG BASED MSGPTR TH$MSG;
    DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;

    ECHO$CHAR:
      PROCEDURE (CHAR);
        DECLARE CHAR BYTE;
        INTMSG = RQWAIT(.RQL7EX,0);
        OUTPUT(USART$OUT) = CHAR;
      END ECHO$CHAR;

    CALL INITIALIZATION;

    CALL RQELVL(6);
    CALL RQELVL(7);

    DO FOREVER;
      MSGPTR = RQWAIT(.RQINPX,0);
      BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
      PTR = 0;
      CHAR = NOT CR;
      DO WHILE CHAR <> CR;
        INTMSG = RQWAIT(.RQL6EX,0);
        IF (CHAR := INPUT(USART$IN) AND 7FH) = RUBOUT THEN
        DO;
          IF PTR = 0 THEN
            CALL ECHO$CHAR(BELL);
          ELSE
          DO;
            CALL ECHO$CHAR(BUF(PTR));
            PTR = PTR - 1;
          END;
        END;
        ELSE
        DO;
          IF CHAR = CONTROL$X THEN
          DO;
            CALL ECHO$CHAR('#');
            CALL ECHO$CHAR(CR);
            CALL ECHO$CHAR(LF);
            PTR = 0;
          END;
          ELSE
          DO;
```

```
6               IF CHAR = CONTROL$R THEN
6               DO;
7                 CALL ECHO$CHAR(CR);
7                 CALL ECHO$CHAR(LF);
7                 DO I = 1 TO PTR;
8                   CALL ECHO$CHAR(BUF(I));
8                 END;
7               END;
                ELSE
6               DO;
7                 IF PTR < MSG.COUNT THEN
7                   BUF(PTR := PTR+1) = CHAR;
                  ELSE
7                 DO;
8                   IF CHAR <> CR THEN
8                     CHAR = BELL;
8                 END;
7                 IF CHAR = ESC THEN
7                 DO;
8                   CALL ECHO$CHAR('$');
8                   CHAR = CR;
8                 END;
7                 CALL ECHO$CHAR(CHAR);
7               END;
6             END;
5           END;
4         END;
3         IF PTR < MSG.COUNT THEN
3           BUF(PTR:=PTR+1) = LF;
3         MSG.ACTUAL = PTR;
3         MSG.STATUS = 0;
3         CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
3         CALL ECHO$CHAR(LF);
3       END;
2     END RD$MIN;

1   WR$MIN:
      PROCEDURE PUBLIC;
2       DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
2       DECLARE PTR BYTE;
2       DECLARE MSG BASED MSGPTR TH$MSG;
2       DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;

2       CALL RQELVL(7);

2       DO FOREVER;
3         MSGPTR = RQWAIT(.RQOUTX,0);
3         BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
3         DO PTR = 1 TO MSG.COUNT;
4           INTMSG = RQWAIT(.RQL7EX,0);
4           OUTPUT(USART$OUT) = BUF(PTR);
4         END;
3         MSG.ACTUAL = MSG.COUNT;
3         MSG.STATUS = 0;
3         CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
3       END;
2     END WR$MIN;
1   END MINIMAL$TERMINAL$HANDLER;
```

APPENDIX B

WRMIN ASSEMBLY LANGUAGE LISTING

```
LOC   OBJ          SEQ        SOURCE STATEMENT

                    1          NAME     WRMIN
                    2          EXTRN    RQELVL,RQOUTX,RQWAIT,RQSEND
                    3          PUBLIC   WRMIN,RQL7EX
00EC                4 DATOUT   EQU      0ECH     ; USART OUTPUT PORT ADR
                    5          CSEG
                    6 WRMIN:
0000 0E07           7          MVI      C,7
0002 CD0000     E   8          CALL     RQELVL   ; ENABLE INTERRUPT LVL 7
                    9 WR0:
0005 110000        10          LXI      D,0
0008 010000     E  11          LXI      B,RQOUTX
000B CD0000     E  12          CALL     RQWAIT   ; WAIT FOR OUTPUT RQST
000E E5            13          PUSH     H        ; PUSH MESSAGE ADDRESS
000F 110700        14          LXI      D,7
0012 19            15          DAD      D
0013 4E            16          MOV      C,M      ; GET RESPONSE EXCHANGE
0014 23            17          INX      H
0015 46            18          MOV      B,M
0016 23            19          INX      H
0017 C5            20          PUSH     B        ; PUSH RESPONSE EXCHANGE
0018 3600          21          MVI      M,0      ; STATUS = 0
001A 23            22          INX      H
001B 3600          23          MVI      M,0
001D 23            24          INX      H
001E 5E            25          MOV      E,M      ; GET BUFFER ADR IN DE
001F 23            26          INX      H
0020 56            27          MOV      D,M
0021 23            28          INX      H
0022 4E            29          MOV      C,M      ; GET COUNT IN BC
0023 23            30          INX      H
0024 46            31          MOV      B,M
0025 23            32          INX      H
0026 71            33          MOV      M,C      ; ACTUAL = COUNT
0027 23            34          INX      H
0028 70            35          MOV      M,B
                   36 WR1:
0029 78            37          MOV      A,B
002A B1            38          ORA      C
002B CA4300     C  39          JZ       WR2      ; EXIT LOOP IF COUNT = 0
002E C5            40          PUSH     B
002F D5            41          PUSH     D
0030 110000        42          LXI      D,0
0033 010000     D  43          LXI      B,RQL7EX
0036 CD0000     E  44          CALL     RQWAIT   ; WAIT FOR TXRDY INTRPT
0039 D1            45          POP      D
003A C1            46          POP      B
003B 1A            47          LDAX     D
003C 13            48          INX      D
003D D3EC          49          OUT      DATOUT   ; TRANSMIT NEXT CHAR
003F 0B            50          DCX      B
0040 C32900     C  51          JMP      WR1
                   52 WR2:
```

```
0043 C1                          POP     B       ; BC = RESPONSE EXCHANGE
0044 D1                          POP     D       ; DE = MSG ADDRESS
0045 CD0000    E                 CALL    RQSEND  ; SEND MSG TO RESP. EXCH
0048 C30500    C                 JMP     WRO
                   ;
                                 DSEG
                   RQL7EX:
000F                             DS      15
                   ;
                                 END
```

APPENDIX C

ATOD PL/M LISTING

```
 1           ATOD:
             DO;

             $INCLUDE(:F1:COMMON.ELT)
 2     1  =  DECLARE TRUE LITERALLY 'OFFH';
 3     1  =  DECLARE FALSE LITERALLY '00H';
 4     1  =  DECLARE BOOLEAN LITERALLY 'BYTE';
 5     1  =  DECLARE FOREVER LITERALLY 'WHILE 1';

             $INCLUDE(:F1:EXCH.ELT)
 6     1  =  DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          =   MESSAGE$HEAD ADDRESS,
          =   MESSAGE$TAIL ADDRESS,
          =   TASK$HEAD ADDRESS,
          =   TASK$TAIL ADDRESS,
          =   EXCHANGE$LINK ADDRESS)';

             $INCLUDE(:F1:IED.ELT)
 7     1  =  DECLARE INT$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)';

             $INCLUDE(:F1:MSG.ELT)
 8     1  =  DECLARE MSG$HDR LITERALLY '
          =   LINK ADDRESS,
          =   LENGTH ADDRESS,
          =   TYPE BYTE,
          =   HOME$EXCHANGE ADDRESS,
          =   RESPONSE$EXCHANGE ADDRESS';
          =
 9     1  =  DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
          =   MSG$HDR,
          =   REMAINDER(1) BYTE)';

             $INCLUDE(:F1:INTRPT.EXT)
10     1  =  RQENDI:
          =    PROCEDURE EXTERNAL;
          =
11     2  =    END RQENDI;
          =
12     1  =  RQELVL:
          =    PROCEDURE (LEVEL) EXTERNAL;
13     2  =      DECLARE LEVEL BYTE;
          =
14     2  =    END RQELVL;
```

```
                =
  15     1      =     RQDLVL:
                =       PROCEDURE (LEVEL) EXTERNAL;
  16     2      =         DECLARE LEVEL BYTE;
                =
  17     2      =       END RQDLVL;
                =
  18     1      =     RQSETV:
                =       PROCEDURE (PROC,LEVEL) EXTERNAL;
  19     2      =         DECLARE PROC ADDRESS;
  20     2      =         DECLARE LEVEL BYTE;
                =
  21     2      =       END RQSETV;

                      $INCLUDE(:F1:SYNCH.EXT)
  22     1      =     RQSEND:
                =       PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
  23     2      =         DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
                =
  24     2      =       END RQSEND;
                =
  25     1      =     RQWAIT:
                =       PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
  26     2      =         DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
                =
  27     2      =       END RQWAIT;
                =
  28     1      =     RQACPT:
                =       PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
  29     2      =         DECLARE EXCHANGE$POINTER ADDRESS;
                =
  30     2      =       END RQACPT;
                =
  31     1      =     RQISND:
                =       PROCEDURE (IED$PTR) EXTERNAL;
  32     2      =         DECLARE IED$PTR ADDRESS;
                =
  33     2      =       END RQISND;


                      /*
                          SBC 711 ANALOG TO DIGITAL BOARD
                      */
  34     1            DECLARE ADC$BASE ADDRESS AT (0F700H);
  35     1            DECLARE COMMAND$REGISTER BYTE AT (.ADC$BASE+0);
  36     1            DECLARE STATUS$REGISTER BYTE AT (.ADC$BASE+0);
  37     1            DECLARE FIRST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+1);
  38     1            DECLARE LAST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+2);
  39     1            DECLARE CLEAR$INTERRUPT$REQUEST BYTE AT (.ADC$BASE+3);
  40     1            DECLARE ADC$DATA$REGISTER ADDRESS AT (.ADC$BASE+4);

  41     1            DECLARE GO$BIT LITERALLY '1';
  42     1            DECLARE AUTO$INCREMENT$ENABLE LITERALLY '2';
  43     1            DECLARE BUSY LITERALLY '8';
  44     1            DECLARE EOS$INTERRUPT$ENABLE LITERALLY '10H';
  45     1            DECLARE EOC$INTERRUPT$ENABLE LITERALLY '20H';
  46     1            DECLARE END$OF$SCAN LITERALLY '40H';
```

```
1      DECLARE END$OF$CONVERSION LITERALLY '80H';

1      DECLARE DUMMY EXCHANGE$DESCRIPTOR PUBLIC;
1      DECLARE RET$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
1      DECLARE GO$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
1      DECLARE TRIGGER EXCHANGE$DESCRIPTOR PUBLIC;
1      DECLARE RET$TRIG EXCHANGE$DESCRIPTOR PUBLIC;

1      DECLARE RQL2EX INT$EXCHANGE$DESCRIPTOR;

1      DECLARE T$PARAM$LOCK EXCHANGE$DESCRIPTOR EXTERNAL;

1      DECLARE BOX$PARAMS(5) STRUCTURE(
               CHANNEL BYTE,
               SET$POINT ADDRESS,
               ERROR ADDRESS,
               OFFSET ADDRESS,
               SAMPLES ADDRESS,
               COUNT ADDRESS,
               ACCUM ADDRESS,
               READING ADDRESS ) EXTERNAL;

1      TICKER$TASK:
         PROCEDURE PUBLIC;
2          DECLARE MSG ADDRESS;
2          DECLARE PULSE$MSG(2) STRUCTURE (
                   MSG$HDR );

2          PULSE$MSG(0).LENGTH,
           PULSE$MSG(1).LENGTH = SIZE(PULSE$MSG(0));
2          PULSE$MSG(0).TYPE,
           PULSE$MSG(1).TYPE = 65;
2          CALL RQSEND(.RET$PULSE,.PULSE$MSG(0));
2          CALL RQSEND(.RET$PULSE,.PULSE$MSG(1));

2          DO FOREVER;
3            MSG = RQWAIT(.DUMMY,5);
3            MSG = RQWAIT(.RET$PULSE,0);
3            CALL RQSEND(.GO$PULSE,MSG);
3          END;

2        END TICKER$TASK;

1      ADC$TASK:
         PROCEDURE PUBLIC;
2          DECLARE TRIGGER$MSG STRUCTURE (
                   MSG$HDR );
2          DECLARE (T$MSG,MSG,LOCK$MSG) ADDRESS;
2          DECLARE I BYTE;
2          DECLARE GAIN LITERALLY '00';
2          DECLARE N$CHNLS LITERALLY '5';

2          TRIGGER$MSG.LENGTH = SIZE(TRIGGER$MSG);
2          TRIGGER$MSG.TYPE = 65;
2          CALL RQSEND(.RET$TRIG,.TRIGGER$MSG);
2          CALL RQELVL(2);
```

```
2      DO FOREVER;
3        MSG = RQWAIT(.GO$PULSE,0);
3        CALL RQSEND(.RET$PULSE,MSG);
3        LOCK$MSG = RQWAIT(.T$PARAM$LOCK,0);
3        DO I = 0 TO N$CHNLS-1;
4          FIRST$CHANNEL$REGISTER = BOX$PARAMS(I).CHANNEL
                            + ROL(GAIN,6);
4          COMMAND$REGISTER = GO$BIT
                      OR EOC$INTERRUPT$ENABLE;
4          MSG = RQWAIT(.RQL2EX,0);
4          COMMAND$REGISTER = 0;
4          BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).ACCUM
                         + ADC$DATA$REGISTER;
4        END;
3        CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
3        T$MSG = RQWAIT(.RET$TRIG,0);
3        CALL RQSEND(.TRIGGER,T$MSG);
3      END;

2    END ADC$TASK;

1   CONTROL$TASK:
     PROCEDURE PUBLIC;
2      DECLARE (LOCK$MSG,T,MSG) ADDRESS;
2      DECLARE I BYTE;
2      DECLARE NCHNLS LITERALLY '5';
2      DECLARE TURN$LAMP$ON
         LITERALLY 'OUTPUT(0E7H)=SHL(I,1)';
2      DECLARE TURN$LAMP$OFF
         LITERALLY 'OUTPUT(0E7H)=SHL(I,1)+1';
2      DECLARE SETUP$8255 LITERALLY 'OUTPUT(0E7H)=80H;
                                  OUTPUT(0E6H)=0FFH';

2      SETUP$8255;

2      DO FOREVER;
3        MSG = RQWAIT(.TRIGGER,0);
3        CALL RQSEND(.RET$TRIG,MSG);
3        LOCK$MSG= RQWAIT(.T$PARAM$LOCK,0);
3        DO I = 0 TO NCHNLS-1;
4          BOX$PARAMS(I).COUNT = BOX$PARAMS(I).COUNT + 1;
4          IF BOX$PARAMS(I).COUNT
             = BOX$PARAMS(I).SAMPLES THEN
4          DO;
5            T,
             BOX$PARAMS(I).READING
               = (BOX$PARAMS(I).ACCUM
                 /BOX$PARAMS(I).SAMPLES) / 38
                 + BOX$PARAMS(I).OFFSET;
5            IF T <= BOX$PARAMS(I).SET$POINT
                     - BOX$PARAMS(I).ERROR THEN
5              TURN$LAMP$ON;
             ELSE
5              IF T >= BOX$PARAMS(I).SET$POINT
                       + BOX$PARAMS(I).ERROR THEN
```

```
116    5                  TURN$LAMP$OFF;
                      BOX$PARAMS(I).ACCUM,
                      BOX$PARAMS(I).COUNT = 0;
118    5            END;
119    4          END;
120    3          CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
121    3        END;

122    2      END CONTROL$TASK;

123    1    END ATOD;
```