Real-time executives for microprocessors

Frits van der Linden and Ian Wilson review the concepts of real-time executives and compare three commercially available executives for 8080/5 and Z80 systems

Principles of real-time executives for microcomputer systems are discussed, together with some secondary functions. Salient features and limitations of three commercially available executives for 8080/5 and Z80 systems are described. An example is given illustrating the use of an executive in a multitasking application involving a simple data logger with a high priority data acquisition task, a low priority data converting task and storage task.

Real-time systems can often be analysed into a set of loosely coupled processes (tasks), each responsible for a single function or a group of related functions. In order to service and control more than one external device, the control system must be capable of running parallel tasks concurrently. With a single processor, true concurrency is evidently impossible as the processor can only execute one instruction at a time. Some mechanism for allocating processor time to tasks, and controlling the execution and synchronization of the tasks is required. In addition, most real-time systems are required to handle asynchronous I/O and respond to time-related events.

In very small systems the controlling logic will be distributed throughout the program, but in a system comprising more than three or four tasks, the system control functions are best integrated in an executive. Tasks communicate with each other and with the outside world by means of requests to the executive: the overall effect is to provide each task with a similar protected environment: the virtual machine. In accordance with this concept, resources such as CPU time can be shared by several tasks, and each task appears to have control of the shared resource. The executive allocates CPU time on the basis of the relative importance, the priority, of the task. Tasks' requests for nonshareable resources such as a peripheral are serviced by the executive on the basis of first in, first out (FIFO). Hardware interrupts are treated as messages to or from peripheral devices for which a task can wait, as if the interrupt were a message from another task. Management of these activities is the primary function of the executive.

PRIMARY FUNCTIONS OF THE EXECUTIVE

An environment capable of supporting multiple real-time tasks must provide the following nucleus of functions (see Figure 1):

BSO/AT Utrecht, Wilhelminalaan 5, Postbus 3059, 3502GB Utrecht, Holland

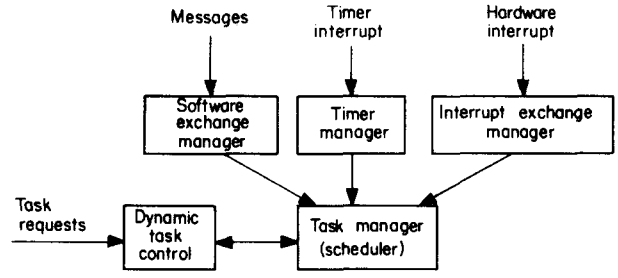

Figure 1. Nucleus of a real-time executive

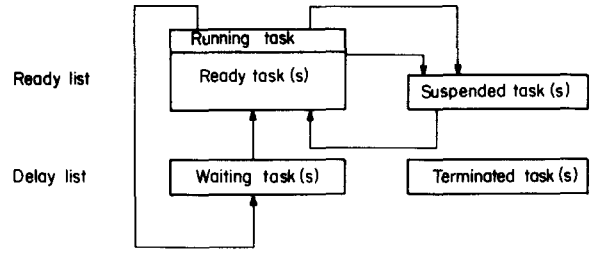

Figure 2. Task states

- task initiation and (re)scheduling (context control)
- intertask communication and synchronization
- timing
- interrupt handling
- dynamic task control

Context control

Each real-time task has an associated prioity and, at any instant, a state known to the executive. In the discussion below, we refer to the following task states (see Figure 2):

- ready
- waiting
- suspended

A ready task is one with no outstanding executive requests. It will be given control of the CPU when no higher priority tasks are ready. A waiting task requires the occurence of an event before it is readied. The event may be external (e.g. an interrupt) or internal (e.g. a message from another task). The running task is the highest priority ready task. The suspended task is a task that is not waiting or ready or competing for any system resource. It is the responsibility of the executive to maintain the state of the tasks, and to reschedule task activity whenever something significant happens.

 0141/9331/80/060211–08 $02.00

Rescheduling (a context switch from the current running task) occurs when one of the following takes place:

- running task waits for a message or time delay
- a higher priority task is readied by an interrupt or a timeout
- running task sends a message to a higher priority task which is thereby readied

The rescheduling process is transparent to the task: the state of its 'virtual machine' is restored when the task resumes execution.

The executives discussed in this paper maintain a ready list, containing references to ready tasks in order of priority: the running task is taken from the top of the ready list. A system-provided idle task is usually placed at the end of the ready list. This task runs whenever all the user tasks are waiting for something (which should be a frequent occurence in a well designed system).

Communication and synchronization

Tasks communicate by means of messages, transmission and control mechanisms for these messages being supplied by the executive. The messages themselves are not transmitted when a task requests the executive send-message primitive: instead, the executive deals with pointers. This method minimizes overhead, but requires a certain amount of discipline: once a message has been sent, its contents should not be modified until acknowledgement of receipt has been obtained. Rather than require the sending task to direct its message to a specific receiving task, the executives considered here maintain system nodes ('exchanges' or 'channels') where messages are queued on a FIFO basis. It is also possible that a number of tasks wait at an exchange for a message to arrive. A message arriving at this exchange will be accepted by the first task in the task queue. It is never possible for tasks and messages to be queued at an exchange at the same time. This communication method, with its built-in buffering, is an extremely powerful tool for real-time systems, as the example in the second part of this article illustrates.

Intertask messages may be grouped into two classes (the 'virtual' analogues of I/O data and flags):

- messages whose content is most important
- messages whose occurence is most important

The first class of message is used when tasks produce and consume data: for example, a terminal handler may build a string of characters entered from the keyboard and then transmit a line as a message to a command interpreter task.

The second class of message finds application where two tasks require synchronization, and most importantly, where a mutual exclusion mechanism is required. Mutual exclusion is necessary whenever two or more tasks may compete for a nonshareable resource. An important example is a system with floating-point maths hardware: while it may be possible to make reentrant drivers, this may sacrifice most of the performance offered by the hardware. Instead, any task requiring use of the resource waits until it is granted exclusive access to that resource: it then performs a burst of activity before relinquishing the resource. Also, when two or more tasks of different priority have access to some global data, a 'lock out' mechanism is required to avoid a low priority task updating the data being 'preempted' by a high priority task also requiring access to the data. This exclusive access scheme is implemented as follows. Each such resource is associated with a single exchange. During system initialization, a single (token) message is queued at this exchange. A task requiring use of the resource then waits at this exchange for the token. This is held while the task uses the resource, thus locking out other tasks competing for the resource, and is sent back to the exchange when the task has completed its current burst activity. Note that through this arrangement, a higher priority task taking over control of the CPU is still excluded from using the resource. However, when two or more tasks require access to a number of non-shareable resources they may become locked in 'deadly embrace'[8,9]: the first task holds some of the resources and is waiting for the others which are held by a second task waiting for the resources held by the first task. Avoiding this potentially disastrous problem may require rigorous design discipline.

Timing

Some form of hardware timer is required to provide delay functions in real-time systems, for two main reasons:

- software timing loops are defeated by interrupts
- a high priority task in busy-wait, locks out all lower priority tasks, which could otherwise be usefully employed

The executives discussed in this paper require a repetitive interrupt for timing purposes. The choice of the period between timing interrupts (the 'tick' time) has to be made carefully. The application tasks require as short a tick as possible, in order to specify elapsed times accurately. On the other hand, the executive has to use some CPU time each tick: if this time is T_e, and the tick period is T, then the fraction of CPU capability available to the user tasks is

$$1 - \frac{T_e}{T}$$

As the clock frequency increases, the available processing power decreases. Figures for T_e are rarely found in the executive specifications. The tick period is normally in the range 10–100 ms. The main timing function provided is timed-wait, given in units of ticks. The executive maintains a delay list, similar to the ready list, with tasks linked in order of relative delay (in this way, the stored time intervals remain constant as tasks whose delay periods have been satisfied are removed from the list). A frequent, and very useful enhancement is the 'wait for time period or message' function which provides a timeout mechanism.

Interrupt handling

Normally, interrupts are intercepted by the executive, which translates the interrupt into a virtual I/O message queued at an interrupt exchange to be handled by a task as for any other message (the task may have special 'interrupt task' attributes). The task fielding the interrupt message will normally invoke a lower priority task and then return to its wait state (to maximize throughput). The disadvantage of allowing the executive to service interrupts is in the response time achieved: the executive has to save the context of the interrupted task and schedule the interrupt task. The response time may be

many hundreds of microseconds rather than the few tens of microseconds typical of a 'naked' interrupt service routine. To overcome this, it is usually possible to override the executive interrupt-handling mechanism and provide an interrupt routine which operates outside the executive framework. After servicing the interrupt, the routine is required to give the executive an acknowledgement before exiting. The normal executive interrupt handling mechanisms are arranged in such a way that the interrupt control hardware of the target system fits nicely into the priority scheme already established. Thus, the asynchronous interface to real events has been integrated to form a coherent structure, rather than being an *ad hoc* addition to the system.

Dynamic task control

In complex systems it is useful for tasks to be able to control other tasks. Tasks may be

- created (made known to the executive and placed on the ready list)
- suspended (rendered inactive temporarily)
- resumed (activation of a suspended task)
- killed (rendered permanently inactive)

Suspension and resumption of tasks are the mechanisms used by the debug task for interactive control. These functions are also useful in systems which may be decomposed into well defined states with known transition paths.

Running tasks can also modify the system configuration by creating or deleting exchanges. Dynamic system configuration alterations are most often used when a library function is added to the application tasks. A library function usually consists of more than one task and a number of exchanges but these are dynamically created and the user specifies only the master task when configuring his system.

SECONDARY FUNCTIONS OF THE EXECUTIVE

Commercial executives usually provide facilities over and above the minimal set described above, to support the development environment, the application, or both. These secondary functions are implemented as tasks, activated by requests from user tasks. Important secondary functions are (see Figure 3):

- debugger
- free space management
- peripheral support
- real-time clock
- bootstrap loader

Debugger

Because of the complex and concurrent nature of real-time programs, the traditional static methods used to debug microcomputer programs (execution breakpoints and trace, register and memory examination and modification) are of very limited usefulness. Instead, dynamic methods are required, providing a window into the running system, and allowing interactive manipulation of the real-time environment. Real-time executives usually provide some form of debugger which may be configured into the prototype system, and omitted from the final production system. In the development environment, the debugger is useful not only in the true debugging phase (getting the system to run) but also for tuning the system (assigning priorities, stack allocation, etc.) The two most important functions of the debugger are:

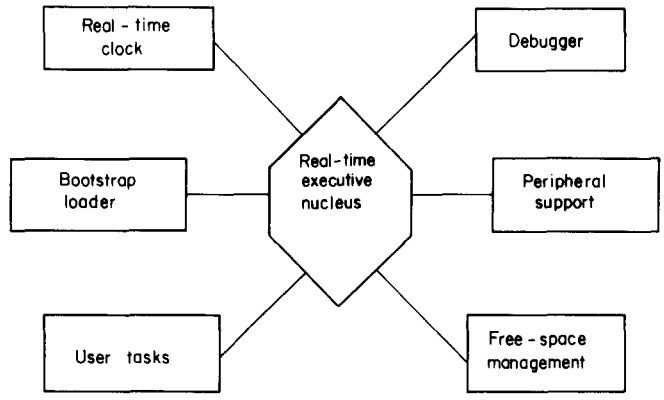

Figure 3. Secondary functions of the executive

- activating and suspending tasks
- visualizing the system data structures

The first of these functions can, with careful planning, allow the real-time system to be built up incrementally. If the system is arranged so that only trusted tasks are activated during the initialization process, then the debugger can be used to incorporate the remaining tasks, one at a time, while system function is monitored. The second main debugger function allows the state of the system to be conveyed to the user in a readable form. Thus, on request, a snapshot of the queue at an exchange, or the state of a particular task, or the contents of a message, may be conveniently formatted and displayed.

The debugger will usually share the system console, being activated by a control character from the console input. There is normally a mechanism for locking out task console output. Other common debug functions include dynamic execution and access breakpointing, and stack monitoring.

One point must be noted. The debugger is implemented as a task, and thus perturbs the real-time environment. Care must be taken to ensure that the application system performs in the same way as the development model under worst-case conditions.

Free-space management

Real-time systems characteristically contain tasks whose main function is to fill up a buffer and then pass it on to a processing task, or to consume a buffer and then wait for more. Due to the nondeterministic nature of the system, it is nearly impossible to determine the memory requirements at any moment. Particularly, when a producing task during short bursts of activity fills up a buffer much faster than the consuming task can process them, multiple-buffering is required. If the worst-case requirements of all tasks can be met, then a static allocation can be made. In the majority of microcomputer applications, however, this is impractical, and the system RAM becomes a shareable resource.

The free-space manager provides a mechanism for allocation of this resource to tasks. Now, instead of a producer task owning a buffer space, it requests memory space from the free-space manager for buffers when it needs them. (The physical memory space is static, of course: in common with other executive functions, the free-space manager only supplies a pointer to a

buffer.) Once the contents of the buffer have been processed by the appropriate consumer task, the buffer space is returned to the free-space manager which concatenates all returned blocks into contiguous memory to reduce fragmentation.

It should be noted that use of the free-space manager does not alter the statistics of the application environment. Careful consideration must be given to peak loading and required throughput, and the system should be designed with wide margins of safety.

Peripheral support

Most microcomputer systems support some form of console I/O for operator communication. The executive may provide (configurable) tasks to handle terminal functions such as line editing, type-ahead, etc. Even if not required in the target system, these utilities are valuable during the development phase (particularly to allow a debugger to be used). Some executives provide management functions for disc support, at the level of logical files. Basic functions such as create, delete, open, close, are provided, together with mechanisms for properly controlled access in the real-time environment and facilities for overlaying programs. Facilities for analogue I/O and for management of hardware floating point are also common.

Real-time clock

Most of the executives provide a very low level real-time clock facility. Normally, time of day is updated from the hardware clock interrupt routine. The time of day may be set and read, but tasks cannot be scheduled for activation or suspension at a specified time.

Bootstrap loader

In some applications it may be desirable to store the program on disc rather than in ROM. A modification to the application program then involves changing a disc file rather than reprogramming the PROM. It is also feasible to use the same hardware for different applications and to store all programs on disc. The bootstrap loader is used to bring the required program into RAM. The nucleus of the real-time executive with the bootstrap loader and part of the disc manager must, however, reside in ROM.

EXECUTIVE-RELATED DATA STRUCTURES

Executives use several control structures for task and system management. In addition, tasks themselves have access to the system nodes used for communication and synchronization by means of task-generated messages.

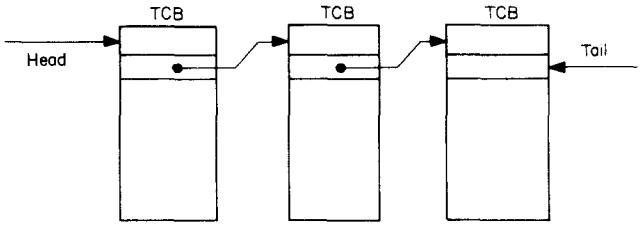

Figure 4. Ready list with three tasks

The significance and general content of these various structures are given briefly below.

Task control block (TCB)

This is the RAM area where the executive maintains the information giving the task's current state. This information will include the task program counter, private stack pointer, priority, status (ready, waiting, etc.), link fields for threading tasks into ready, wait or delay lists (see below), and some form of task identifier for debug purposes.

Static task descriptor

This is the user-provided template from which a task's TCB is constructed during system initialisation. It will contain the initial program counter, stack pointer, and priority for the task, and may have other information such as stack length and task name in ASCII for debugging purposes.

Ready list (see Figure 4)

This is a list of ready tasks, ordered by priority. The executive keeps track only of the head and tail tasks of the list, the link field of the TCB being used to thread tasks together. (The information in the link field is the address of the next task's TCB.)

Delay list

This list contains waiting tasks, threaded in the same way as described for the ready list. Tasks are ordered by relative delay. The executive keeps only a pointer to the head of the list, and the number of ticks until the head task is to be awakened.

Exchange control block

Exchanges are the nodes where tasks and messages are associated by the executive. Whenever there is a surplus of tasks, or messages, they are linked into a list using their link fields. This time, the list ordering is in order of arrival, giving the required FIFO mechanism. Even though only tasks or messages can be queued at any one exchange, it is normally simpler to maintain pointers to both queues at an exchange.

Message control block

These are the structures most relevant to the tasks. One or more fields in the message are reserved for executive use only, for threading messages in an exchange queue. The remainder of the message is relevant to the generating and receiving task. As messages are usually the sole means of intertask communication in a well structured system, it is important to define protocols for their structure. Often there is a defacto standard employed by utility tasks supplied with the executive (e.g. terminal handler). In this case, it is advisable to adopt this standard for use with the application tasks. Most practical protocols treat the first part of the message as a header giving message length and type, followed by either the data or a pointer to the data. A good example is the protocol where for short messages (length one to four bytes) the data is contained in the four bytes following the message header: for longer messages, these four bytes contain a buffer pointer and buffer length.

COMMERCIALLY AVAILABLE REAL-TIME EXECUTIVES

There are three real-time executives commercially available for 8080-family microcomputers: RMX/80, RTM8 and REX-80. Minimum system requirements for the executive nuclei are similar (typically 2–3k ROM, 0.5k RAM and a hardware timer/counter), but there are many differences in implementation, configurability and the extensions available. Major features and limitations are discussed below.

RMX/80

Intel's RMX/80[2] was one of the first real-time executives for 8080-based microcomputers. It is designed for the Intel SBC range, and is closely bound to the specific system architecture (there are different RMX/80 versions for the various environments). RMX/80, apart from the architectural dependency noted above, is highly configurable: for example, several primitives typically required only during development (e.g. suspend and resume) may be omitted. All RMX/80 primitives observe the parameter-passing conventions of the high level language PL/M[6]. Furthermore, PL/M provides for operations on data structures for which only a pointer is given ('based' variables). Thus it is natural to write noncritical tasks in PL/M and to use assembler only for time-critical items such as interrupt routines outside the RMX/80 framework.

One of the idiosyncracies of RMS/80 is the WAIT primitive which returns either after a timeout or when a message is available at the exchange concerned. This means that a pure wait-for-message has to specify infinite delay (by specifying a zero delay under RMS/80) and a simple delay can be achieved only by waiting at an exchange where no message is ever sent.

RMS/80 is supplied together with a terminal handler and debugger. The latter, when configured in its active form (which allows interaction rather than just observation) is a very powerful tool for system development and timing. There is also a configurable free-space manager.

One of the strengths of RMX/80 lies in the range of secondary functions available. To some extent this reflects the time RMX/80 has been available and the level of investment involved, but it also emphasis the ready portability of tasks given a well defined message protocol. In addition to the terminal handler tasks already mentioned (configurable for a wide range of options), there is an analogue handler, floating point hardware manager, a disc file handler, and a bootstrap loader.

Also available are a BASIC run-time package based on RMX/80 which provides the user with a resident programming language, with a FORTRAN run-time package for number crunching applications.

In conclusion, RMX/80 offers a flexible, well supported nucleus, with configurable extensions, for real-time systems. Use is restricted (by licence) to Intel SBCs with 8080 and 8085 microprocessors.

RTM8

RTM8[4] is a straightforward executive with no frills designed for the AMC 95/4000 series Monoboard computers. It provides most primary functions, but secondary functions only include a real-time clock, a file manager, a (static) debugger, and a terminal handler.

RTM8 intercepts all interrupts and may perform on behalf of a task the inbound or outboard transfer. Byte I/O in fixed or variable format, ASCII input with echo are supported.

System configuration is performed by the highest priority task which gets control after system initialization by RTM8. This task creates all other tasks and channels. In accordance with this, tasks can set the priority of other tasks.

Particularly cumbersome in RTM8 is passing parameters to and returning results from system primitives. Before a system call can be made the task must set up a block containing the function code, and parameters required by the system primitive. The result of a primitive invocation is written again into this parameter block. (In other executives parameters and results are passed through the internal registers and stack.)

In conclusion, RTM8 is an easy-to-use executive, which offers sufficient functions for simple real-time applications. The absence of a free-space manager, the parameter block convention for system primitive calls, and the purely static debugger are serious disadvantages. An advantage is that the user does not have to provide interrupt service routines or interrupt tasks.

REX-80

REX-80[3,5] has recently appeared on the market and has been developed by an independent software house. This executive has several features not found in other executives.

The system nodes where messages are queued are called channels. Two types of channel exist: interrupt and software channels. Any task may send an interrupt message to an interrupt channel, requesting an inbound or outbound transfer. Associated with each interrupt channel are four user-written routines for initializing the transfer, handling each interrupt, terminating the transfer, and error handling. In addition to a channel control block, REX-80 requires a static channel descriptor providing pointers to these routines. The initialization routine enables the interrupt and initializes the device. The interrupt handler receives from REX-80 a pointer to the interrupt channel and the interrupt message containing information on type of transfer, buffer to be used, etc. This routine does the actual transfer and signals completion to REX-80. The terminating routine invoked by REX-80 disables the particular interrupt. After transfer termination, the requesting task is informed by means of its own interrupt message whether the transfer was successful. The main advantage of REX-80 interrupt handling is that it does not require a full context switch at every interrupt.

Unsolicited interrupts, i.e. interrupts for which no task is waiting at an interrupt exchange, should normally be avoided, but can be handled within the framework of REX-80.

Task synchronization with other tasks or I/O may occur at 'message hand-off' or some time after this. In the former case the sending task will wait until the receiving task accepts the message. The highest priority task then gets control of the processor. In the case that synchronization occurs some time after 'message hand-off', the sending task associates one of its 16 event flags, the 'virtual interrupts',

with the 'message hand-off'. After sending the message the task continues processing if it is the highest priority task. At some time the task may wait to synchronize with the receiving task, and will wait for the event flag to change state. The message receiving task, when it is running, signals the sending task by means of the event flag at some stage during processing that the message has been accepted. The highest priority task then continues processing.

The event facility in REX-80 is potentially very powerful and avoids introducing acknowledge channels (as in RMX/80). Unfortunately, REX-80 cannot distinguish between events that never have been requested or have already occured.

REX-80 maintains a real-time clock and a time-of-day. Tasks can be delayed for certain time periods and also the future occurence of an event can be timed out. This is not achieved by waiting at dummy channels but by the MARK TIME AND WAIT primitive. Furthermore, REX-80 provides an elegant primitive for waiting for the first of a number of events to occur.

Dynamic task control in REX-80 is very limited: it allows tasks and channels to be created dynamically, but only channels to be deleted. A task can only suspend itself. A useful feature of REX-80 is that a running task may alter its priority in case it does not want to be preempted by other tasks during critical operations.

REX-80 is available for Z80 and 8080/8085-based systems. As the source is distributed as well, the nucleus of the executive can be tailored to the application. The user decides which I/O addresses to allocate to the interrupt controller, timers, etc. and selects the system clock frequency satisfying his requirements. The free-space manager is integrated in the nucleus. It does not provide a facility for a task to wait for space to become available (as the RMX/80 free-space manager does). An interrupt handler for the Am9511 arithmetic processor capable of accepting entire expressions (in reverse polish notation) is a useful addition to REX-80 secondary functions. Other peripheral support include a terminal handler and an analogue I/O handler. REX-80 is not directly PL/M compatible but an interface has been developed by the authors.

In conclusion, the rich REX-80 nucleus offers more flexibility than RMX/80 or RTM8 (at the expense of its complexity). Novice users will find it difficult to select the functions they require for their particular problem. This may be partially due to the rather academic users guide. The absence of a debugger and more peripheral support functions is a serious shortcoming.

EXAMPLE

One of the most common applications of microprocessors is in data logging. A number of I/O ports are connected to sensors and changes in the input values are stored in a data buffer with the time of the day. When the data buffer has been filled, it is dumped on to, for example, a cassette. The data logging function can be broken down into two tasks, a data acquisition and data storage task. This is shown graphically in figure 5 where circles represent channels, rectangles are tasks, and the arrows connecting circles and rectangles, the sending or receiving of messages. The number in the rectangle indicates the task priority (1–7, 7 being highest priority).

The data acquisition task reads the input ports at regular intervals (e.g. 10 system time units) and if the inputs have changed, a message is sent to channel 1 with the new input values and the time of the day. The storage task waits at this channel and when a message arrives the data in the message is converted (e.g. add offsets etc.) and stored in a buffer. If this would cause buffer overflow, the current buffer is emptied on to cassette and the data written into the empty buffer.

Essentially, each task in this example has the following structure (using PL/M):

```
TASK:
        /* declarations */
        DO FOREVER;
            /* body of task */
        END;
END TASK;
```

To implement the acquisition task under REX-80, the following primitives are required:

- MRKTW (DELAY) – marks a time delay in system time units
- SEND (CHANNEL, MESSAGE, EVENT) – send a message to a channel and wait till message has been received

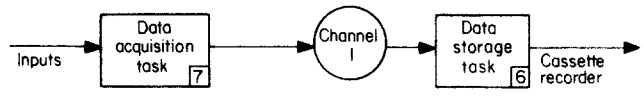

Figure 5. Two-task data logger

```
DECLARE MESSAGE STRUCTURE(
   RESERVED_FOR_REX80(3) ADDRESS,
     /* initialise BUFFER_POINTER with address of MESSAGE.REMAINDER */
   BUFFER_POINTER ADDRESS INITIAL (.MESSAGE.REMAINDER),
     /* TRANSFER_COUNT is initialised to 16 bytes */
   TRANSFER_COUNT ADDRESS INITIAL (16),
   REMAINDER(16) BYTE);
DECLARE EVENT BYTE;

DATA_ACQUISTION_TASK: PROCEDURE PUBLIC;
    DO FOREVER;
       /* wait for 10 system time units */
      CALL MRKTW(10);
       /* read new input values into message; if they have changed, retur
          TRUE, else return FALSE */
      IF READ_INPUTS(.MESSAGE.REMAINDER+3) THEN DO;
           /* get time of day  into message */
          CALL GTIMD(.MESSAGE.REMAINDER);
           /* send the message to channel 1 and associate event 1 */
          CALL SEND(1,.MESSAGE,1);
           /* wait on event 1 indicating that message has been accepted *
          EVENT=WAITE(1);
      END;
    END;
END DATA_ACQUISITION_TASK;

DATA_STORAGE_TASK: PROCEDURE PUBLIC;
    DECLARE PTR ADDRESS;

    DO FOREVER:
       /* wait for message at channel 1 and return with pointer */
      PTR= RECVW(1);
       /* add offsets, etc. */
      CALL CONVERT_DATA(PTR);
       /* add message to buffer and empty buffer onto cassette if full */
      CALL ADD_MESSAGE_TO_BUFFER(PTR);
        /* acknowledge receipt */
      CALL SIGNL(PTR);
    END;
END DATA_STORAGE_TASK;
```

Figure 6. Two-task data logger without buffering

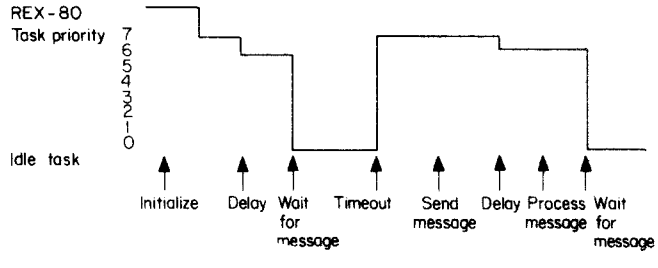

Figure 7. Task interleaving

- RECVW (CHANNEL) – wait at channel for message
- WAITE (EVENT1 OR EVENT2 OR . . .) – wait for event(s) and returns highest priority event number
- GTIMD (POINTER) – fill buffer starting at POINTER with current time-of-day
- SIGNL (POINTER TO MESSAGE) – acknowledge through message, receipt of message

The data logger tasks are declared in the (incomplete) code segment in Figure 6.

The flow of control in this two-task system is as follows (see Figure 7). After power-up, REX-80 initializes various data structures and devices, using a user-provided cold-start routine with descriptors of tasks and channels and initialization code for the devices. The control of the processor is then transferred to the highest priority task, in this case the acquisition task. This task is delayed for 10 system time units by the MRKTW primitive and control is passed on to the storage task. This task checks if there is a message at channel 1 and is made waiting as there is no message available. The idle task gets control until the acquisition task is readied by a time-out. The acquisition task prepares and sends (if necessary) a message

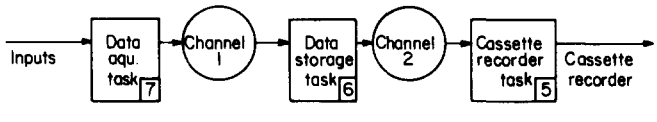

Figure 8. Three-task data logger

```
DATA_ACQUISITION_TASK: PROCEDURE PUBLIC;
    DECLARE PTR ADDRESS;
    DECLARE MESSAGE BASED PTR STRUCTURE(
      RESERVED_FOR_REX80(3) ADDRESS;
      BUFFER_POINTER ADDRESS,
      TRANSFER_COUNT ADDRESS,
      REMAINDER(16) BYTE);
    DECLARE EVENT BYTE;

     /* get space and create message */
    PTR=ALLOC(1);
    MESSAGE.BUFFER_POINTER=.MESSAGE.REMAINDER;
    MESSAGE.TRANSFER_COUNT=16;

    DO FOREVER;
      CALL MRKT(1,10);
      IF READ_INPUTS(.MESSAGE.REMAINDER+3) THEN DO;
          CALL GTIMD(.MESSAGE.REMAINDER);
           /* send message but do not associate event */
          CALL SEND(1,.MESSAGE,0);
           /* create new message */
          PTR=ALLOC(1);
          MESSAGE.BUFFER_POINTER=.MESSAGE.REMAINDER;
          MESSAGE.TRANSFER_COUNT=16;
           /* wait for event 1 */
          EVENT=WAITE(1);
      END;
    END;
END DATA_ACQUISITION_TASK;

STORAGE_TASK: PROCEDURE PUBLIC;
    DECLARE PTR ADDRESS;
    DECLARE BUFPTR ADDRESS;
    DECLARE BUFFER BASED BUFPTR STRUCTURE (
       RESERVED_FOR_REX80(3) ADDRESS,
       POINTER ADDRESS,
       TRANSFER_COUNT ADDRESS,
       REMAINDER(1024) BYTE);

     /* create buffer in message format */
    BUFPTR=ALLOC(16);
    BUFFER.TRANSFER_COUNT=0;
    BUFFER.POINTER=.BUFFER.REMAINDER;

    DO FOREVER;
       PTR=RECVW(1);
       CALL CONVERT_DATA(PTR);
       CALL ADD_MESSAGE_TO_BUFFER(PTR,.BUFFER);
        /* return message to free space manager */
       CALL DALCT(PTR);
       IF BUFFER.TRANSFER_COUNT=1024 THEN DO;
            /* send message but do not associate event */
          CALL SEND(2,.BUFFER,0);
           /* create a new buffer */
          BUFPTR=ALLOC(16);
          BUFFER.TRANSFER_COUNT=0;
          BUFFER.POINTER=.BUFFER.REMAINDER;
       END;
    END;
END STORAGE_TASK;
```

Figure 9. Data logger with multiple buffering

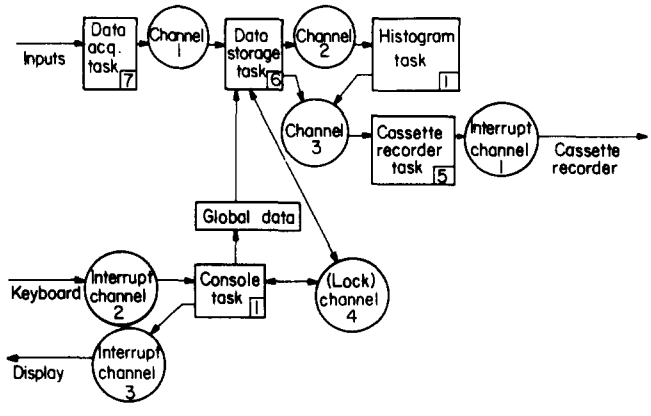

Figure 10. Data logger with console and histogram computation

to channel 1. It then waits until the storage task receives the message. As the acquisition task is the highest priority task, it gets control over the processor.

This simple example illustrates that all scheduling is entirely transparent to the user and, in fact, the two tasks appear to run concurrently.

There are several problems, however, in this implementation. The scan interval is 10 system time units, plus the time it takes to sample the inputs and send a message. The acquisition task must wait for the storage task to receive and process the message as there is only one message available. If the storage task processing takes more than 10 system time units because of dumping the buffer on to cassette, the sample interval could be considerably longer than intended. If the sample interval is critical, queuing of messages to avoid the acquisition task having to wait for the storage task, becomes desirable. In the case of bursts of activity on the inputs and consequently many messages queued at channel 1, and the storage task not able to accept any message because it is waiting for the cassette recorder, data buffer queuing for the cassette recorder may be desirable as well. The diagram in Figure 8 shows a three task, two channel system with multiple buffering at both channels.

To implement this, we need some more REX-80 primitives:

- ALLOC(N) – request to free space manager for N*64 bytes
- DALCT (POINTER) – return 64 bytes to free space manager starting at POINTER
- MRKT (EVENT, N) – mark N system time units interval and associate this with EVENT

The acquisition and storage task with multiple buffering are shown in Figure 9. The cassette controlling task has a similar structure to the storage task and is not shown. Note that the sample interval in the acquisition task is now constant.

It may be desirable to enter the offsets and calibration data, and time-of-day from the console. In parallel to the three tasks in Figure 8 a low priority console interrupt task could interact with the operator. Mutual exclusion techniques must be incorporated to access global data. If more processing time is still available, a low priority task computing histograms can be included (see Figure 10).

This example shows the ease with which new functions can be added to the application and how multiple buffering and pipelining can be achieved using facilities provided by the executive.

CONCLUSION

Many real-time microprocessor applications have broadly similar requirements. The past few years have seen the appearance of several general purpose real-time executives for microcomputer systems. The significance of this can be compared with the appearance of high level language support for microprocessors. There are broadly similar advantages (e.g. speed of program development), disadvantages (e.g. overhead in time and memory space), and tradeoffs to be considered.

The abstractions provided by an executive allows system designers to structure and to implement software for their application without concerning themselves about details of task scheduling, resource allocation, and in effect 'reinventing the real-time wheel'. The software framework of real-time executives could be viewed as a 'software bus' which allows software modules to be 'plugged' into the system. Similar to the hardware bus, it establishes protocols and interconnect paths, and, if properly designed, ensures expansibility.

Modular programming is encouraged as an application is broken down into tasks which can be written by different programmers as independent modules with clearly defined interfaces.

One of the disadvantages of real-time executives is the overhead caused by context-switching. New 16-bit microprocessor architectures have been designed with real-time executive requirements in mind. Context switching by means of segmentation and semaphore operations are facilitated in hardware. It is expected that more functions now provided in software will move to firmware, either microprogrammed or mask-programmed, in order to reduce the ever-increasing software costs.

ACKNOWLEDGEMENTS

The authors are grateful to BSO/AT staff for their valuable comments and fruitful discussions during the preparation of this paper.

REFERENCES

1 **Hansen, P B** 'A keynote address on concurrent programming' *Computer* (May 1979) pp 50–56

2 *RMX/80 user's guide* No 9800522B Intel Corp. (1979)

3 *REX-80 technical manual* Systems & Software Inc., Downers Grove, Illinois, USA

4 *RTM8 real-time executive user's manual* Advanced Micro Computers (1979)

5 **Chien, Y P** 'Multitasking executive simplifies real-time microprocessor system design' *Computer Des.* (January 1979) pp 109–117

6 *PL/M-80 programming manual* No 9800268 Intel Corp. (1979)

7 *RMX/80 real-time multitasking executive* Intel Application Note 33 (1979)

8 **Coffman, E G et al.** 'System deadlocks' *Computing Surv.* Vol 3 (June 1971)

9 **Habermann** 'Prevention of system deadlocks' *CACM* Vol 2 (July 1969) pp 373–377