SPECIAL FEATURE:

# An Architecture for Real-Time Software Systems

**Theodore P. Baker, Florida State University**
**Gregory M. Scallon, Boeing Aerospace**

***The Rex approach to software architecture views executives as independently programmable machines that execute conventional-language application procedures as individual instructions of a higher-level program.***

Rex, an architecture for designing real-time systems based on a programmable standard executive, has evolved from a series of real-time systems in the aerospace industry. Rex's software architecture is characterized by its view of the executive as an independently programmable machine that executes application procedures written in conventional programming languages as if they were individual instructions of a higher level program.

Because it provides a programmable standard interface to compose application program components, the executive lets system designers program in the large. This approach also allows prediction of system performance via prototyping and simulation at the executive level, reuse of software components, flexible use of multiprocessors, and potential for automatic recovery from failures.

A three-tiered view of real-time software architecture underlies Rex. Figure 1 illustrates this view, whose main software system components are

*Application.* The application is specifically designed to solve the user's problem.

*Executive.* The executive provides functions such as module interconnection, scheduling, and dynamic resource allocation via an interface independent of the host hardware configuration. The implementation of the executive may depend on the host but is independent of the application — and is thus reusable between applications on a single machine.

*Operating system.* The operating system provides lower level services to support the executive. The services provided are dependent on the hardware configuration. It is therefore specific to the host machine — but not to the application. Like the executive, it is reusable between applications on a single machine.

(The terms "executive" and "operating system" are used here with special meanings. In particular, the executive is not part of the operating system, even though it performs some traditional operating system functions.)

**Key element.** The key element of this hierarchy is the executive, which effectively defines the interfaces between the application and the operating system levels. A standard executive provides standard interfaces for the other system components. Rex plays this role.

Some of the specific requirements imposed on this standard real-time executive are

- Physical processors can be added without redesigning the applications software or executive.
- Each procedural component is executed in a static environment — that is, insulated from any changes in system state that are not direct effects of its own execution.
- The executive automatically manages the swapping of all nonresident modules (that is, blocks of information such as code and data).
- The executive incorporates control options that automatically adapt scheduling plans to dynamic loading conditions.
- The executive is directly programmed with explicit control parameters.
- All synchronization is accomplished by the executive.

**Two levels.** A basic premise of Rex is that the activities of programming in the large — managing resources and interconnecting components — should be separated cleanly from the activities of programming in the small — programming individual components. Viewed as a virtual machine that treats application program components as defining individual instructions, the executive accomplishes this separation. Programming in the large is


0740-7459/86/0500/0050$01.00 © 1986 IEEE

concerned with producing a program in the machine language of the executive, while programming in the small is concerned with extending its instruction set.

Figure 2 illustrates the separation of these two levels. Application procedures are coded in a conventional programming language and compiled into the machine code of a physical machine. A plan for the management of these procedures' executions to form a system is expressed in a separate system-specification language, and is separately translated into tables (agendas) used by the Rex virtual machine. These tables are in effect a high-level machine code interpreted by the executive.

This machine architecture orientation can be contrasted with that of other work on execution environments for real-time software. Much of this work is oriented toward programming languages (for example, RTL/1,[1] Modula,[2] Iliad,[3,4] and Ada[5]).

Although Rex is designed to support a high-level system-specification language — and can be viewed as defining a high-level machine language — these languages are independent from the language used to program procedural components of the application.

Other work is oriented toward developing operating systems (for example, Thoth,[6] iRMX 86,[7] and VRTX[8]). Unlike these operating systems, which can be viewed as providing a virtual machine for the application program, Rex provides a virtual machine for the system specification.

## The application

The application has three kinds of components:

*Procedural components (procedures).* These perform all the useful work. Each instance of a procedure in a process is called a task.

*Informational components (data sets).* These communicate data forward in time and between tasks.

*Organizational components (processes).* These specify the sequencing of tasks and the interconnection of tasks and data sets.

The overall structure of an application may be described in a system specification language as a collection of data sets and process specifications. The process specifications refer to procedures coded separately in a conventional programming language. A prototype of such a language, Real, has been designed and implemented for use with Rex as part of a simulation-based design tool, FOSS.[9]

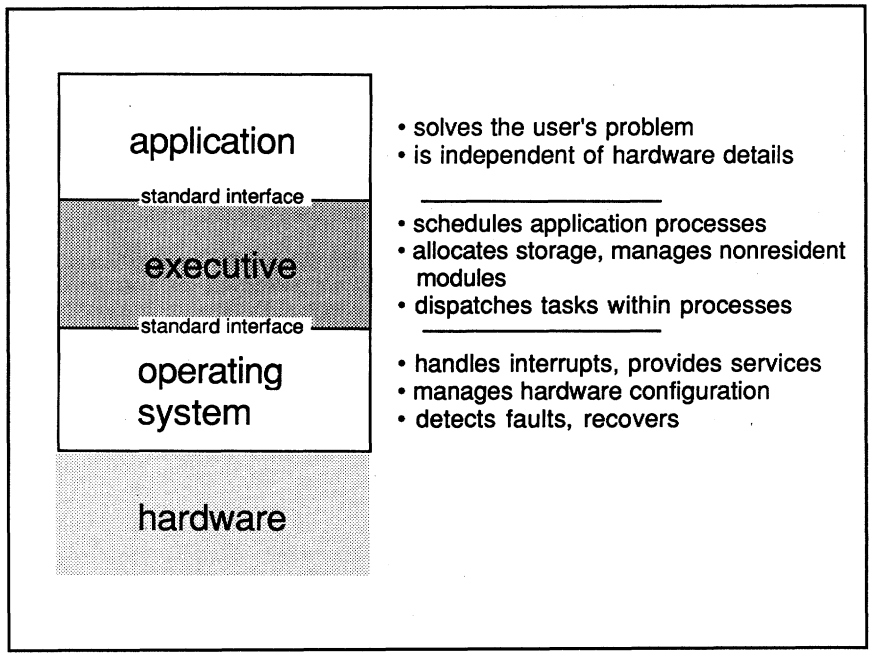



**Figure 1. Real-time software architecture.**

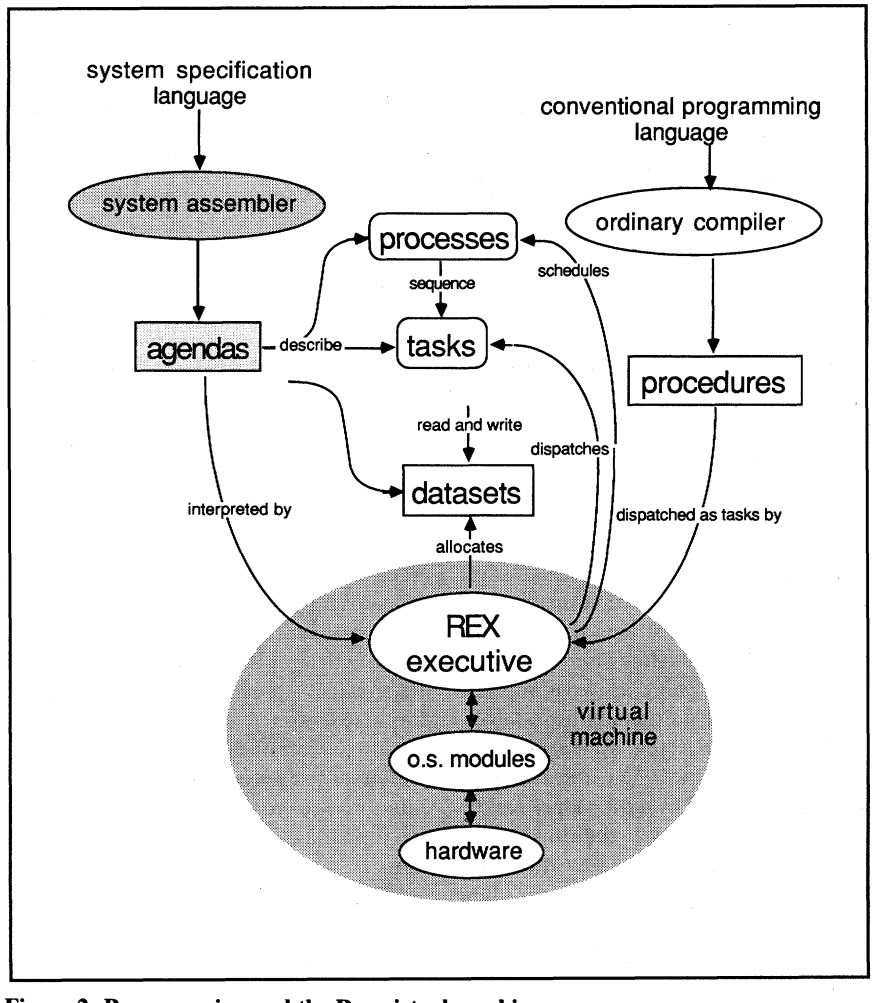



**Figure 2. Programming and the Rex virtual machine.**

# Rex background

Rex is a standard table-driven executive developed at Boeing Aerospace by Greg Scallon. It is the most recent descendant of a series of real-time executives developed for projects in the aerospace industry.

Rex's ideas can be traced back to the AgPrep agenda preparation system developed by Duane Brown Associates for the NASA Apollo program and to IBM's System 360 operating system. Precursors include the executives of the NASA Apollo Range Instrumentation Ship (1965-1970) and the US Army Safeguard ballistic missile defense projects (1970-72).

The Designating Optical Tracking system (1978) successfully uses a multiprocessor-resident implementation of Rex. A simulated nonresident version has also been implemented as part of an ongoing research program on design automation techniques for real-time software systems at Boeing Aerospace.

The goal of this research program is to devise and construct tools a designer can use to produce complex real-time software for large-scale multiprocessor configurations with shared memory and bulk backing storage that supports nonresident modules. The research has focused on developing system specification techniques, performance prediction tools, reusable components, and standard execution environments.

## Further readings


J.A. Grupe and G. Scallon, "Functionally Oriented System Simulation for Computer-Aided Design of Software/Hardware systems," *Proc. Ninth Annual Simulation Symp.*, Tampa, Florida, Mar. 1976, pp. 225-240.

G. Scallon, "Toward Computer-Aided Design of Real-Time systems," *Proc. European Computer Conf. on Software Systems Engineering*, London, Sept. 1976, pp. 59-70.

J.A. Grupe and G. Scallon, "The FOSS User's Guide," Boeing Aerospace Report D180-18617-2, Seattle.

G. Scallon, "Simulation for System Design," Texas Instruments Technical Report U1-834309-12, Process Construction Documentation, Vol. 7.

G. Scallon, "Description of the Functionally Oriented System Simulation (FOSS) for Software/Hardware Design," Boeing Aerospace Report D180-18617-1, Seattle.


Please note that the use of defined terms such as procedure, task, data set, and process in this article is different from common usage elsewhere. In particular, processes and tasks here should not be confused with "processes" in languages such as Mesa, Concurrent Pascal, and Modula, or with "tasks" in languages such as PL/I and Ada. Neither should data sets be confused with notions like files, nor should procedures be confused with "procedures" in any particular programming language.

**Procedural components.** Procedures transform data. For programming in the large, they are viewed as individual tasks. Like instructions of a machine architecture (and the primitive constructs of a programming language), they are reusable components. They may be used more than once, both within a single application system and within different systems.

From an interface view, the transformation performed by a procedure can be abstractly specified as a relation between values of data before and after execution of the procedure. A particular procedure implements a particular algorithm to achieve the specified transformation.

The semantics of procedures are limited in Rex. Each procedure invocation (that is, each task) is viewed as an indivisible unit of work that, once started, is executed without interruption by any other task. Because they are not preemptible, procedures must be simple and have short, bounded execution times. Execution times of some tasks can be longer if a separate processor is reserved for fast-response processes.

Procedures, executed only as tasks, are invoked by the executive. The flow of control between tasks is defined by the organizational components (processes) of the system. These are represented for the executive by tables called agendas. Procedures can share code, and a single procedure may be executed concurrently as different tasks by different processors.

Procedures access all data through explicit parameters. The actual parameters (data sets) of a procedure invocation (task) are passed by reference. Each formal parameter has a specified disposition. These are

*New.* The procedure writes new valid data into the data set. It does not expect any valid data to be in the data set when it starts.

*Unchanged.* The procedure may read data from the data set. It assumes the data set contains valid data. It does not write on the data set.

*Modified.* The procedure may read data from the data set. It may also write new data into the data set. The procedure assumes the data set contains valid data when the procedure starts and guarantees the data set contains (possibly new) valid data when the procedure is done.

*Corrupted.* The procedure may read from and write to the data set. When it is finished, however, the data set may no longer contain valid data. This differs from the disposition "modified" in that the transformation applied to a modified parameter is part of the specified function of the procedure, while any transformation applied to a corrupted data set is not guaranteed to conform to any specification.

Each parameter also has a set of constraints. Constraints are of the kind typically imposed by a data type in a programming language. They include, for example, internal structure, dimensions, and units.

In addition to the formal parameters, a procedure may return a condition code. This is a value that can be used by the dispatcher to control subsequent dispatching via predefined sequence-control tasks.

Since there are no global variables or pointers, no side effects are permitted. Because procedures are only invoked by the dispatcher, aliasing (access to one data set under different names) can also be controlled. There can be no implicit parameters because data may only be contained in data sets, and data sets may only be accessed via formal parameters. Data sets may not contain pointers to other data sets. These restrictions ensure that each procedure's effect is easily understandable in terms of a transformation on explicit parameters.

There are several other reasons to limit procedures. One is to ensure they can be implemented efficiently. Concerns include keeping down the cost of context switches and generating efficient code for the procedures. Limitations on procedures also simplify prototyping and analytical modeling of system designs. The restriction that procedures be reentrant and access all data through explicit parameters makes them sharable at the object code level.

These restrictions, together with the requirement that tasks not be preempted, reduce the need for explicit synchronization of accesses to shared data. They also permit efficient dynamic linking of procedures and efficient management of nonresident modules (that is, code and data sets).

Because all the modules required by a procedure are known to the executive, they can be guaranteed to be in memory before the process invoking the procedure is readied for execution. Nonresident modules may be located anywhere in memory, since their actual addresses are provided to the procedure at the time of invocation via an argument list.

Memory for local temporary data (data that is accessed only by the procedure and that does not persist between invocations) is allocated within a contiguous block (such as a stack frame). The base address of a corresponding block of scratch memory is provided when the procedure is invoked.

In a multiprocessor shared-memory configuration, Rex allocates a fixed block of scratch memory to each processor, allowing the maximum scratch space needed by any task.

More space-efficient strategies have been used for single-processor configurations. In particular, the scratch space required by each task may be treated as a local short-term data set and allocated within the workspace of the process containing the task.

**Informational components.** Data sets provide communication between tasks — between different tasks and between different executions of a single task. They are not expected to be reusable because they are typically designed for a particular application.

***Explicitly specified locks are used to assure the integrity of local data sets.***

A data set has a specified, limited size and a set of attributes. The size and attributes must be compatible with the constraints of any formal parameter it is linked to. A data set may be designated as either a variable or a constant. If it is a constant, it may only be accessed as an unchanged parameter. If it is a variable, it may be accessed as a parameter with any disposition. Other attributes describe features such as format, dimensionality, and structure of the data set's contents.

Data sets may be local or global. Local data sets are used only to communicate between tasks within a single execution of a process. Global data sets are used to communicate between different processes or between processes and the physical I/O of the operating system.

Explicitly specified locks are used to assure the integrity of global data sets. There are several locking options:

*No inhibit.* The data set requires no explicit locking. The access algorithms are self-synchronizing (for example, circular buffers) or there is other design knowledge that solves the data validity/control problems.

*Inhibit concurrent writes.* Only one task at a time may be executed with disposition to write to the data set.

*Inhibit access during write.* No other task that accesses the data set (regardless of disposition) may be executed concurrently with another task with disposition to write.

Two forms of inhibit are provided:

*Blocking wait.* A task is dispatched from some lower priority process. This has the advantage of doing useful work while waiting.

*Busy wait.* The dispatcher idles (that is, dispatches only higher priority processes) until the lock is removed. This may provide faster response in some cases, such as when the lock is held by a task expected to finish soon, or when it is held by an I/O operation expected to finish very quickly. Since a lock can only be held during a single task, the wait can never be longer than the execution time of a single procedure.

Local data sets do not require explicit locks, since they are in effect automatically provided with busy-wait locks through intertask dependency codes.

In Rex, local data sets are further divided into short-term and long-term data sets. Short-term data sets do not persist between executions of a process. They are all allocated within one contiguous block of storage associated with the process they belong to.

This block, the workspace of the process, is allocated dynamically to the process when it is triggered. Short-term data sets whose lifetimes do not overlap can be overlain within the workspace, conserving memory. Because the lifetimes of short-term data sets can be determined from the dispositions of the parameters they are linked to, this overlaying can be determined automatically by a high-level system assembler.

The high-level assembler also determines which local data sets are short-term according to how they are used. (The first task to access a short-term data set does so with disposition "new.") By contrast, long-term data sets (whether local or global) are allocated storage individually.

**Organizational components.** Processes define the system-level structure of a particular application. They specify how the resources of the system are managed to accomplish the desired system performance, including the order in which procedures are invoked to perform tasks and how tasks are linked to data sets. Unlike procedures, processes are not ordinarily expected to be reusable between different applications, since they embody the particulars of their unique applications.

***Processes are run repeatedly at scheduled times or in response to certain events.***

The executive runs a process by interpreting a task list. Each task in the list specifies the procedure that must be invoked to perform the task and the data sets that must be linked to the formal parameters of the procedure.

The task list is one part of an agenda, which is a table that describes a process. Agendas have two other parts. One is the scheduling parameter list that provides information needed by the scheduler, such as the priority and period of the process. The other is the requirement list that specifies resources required by the process, such as special processors, code and data modules, and scratch memory. An agenda is produced by assembling a process specification (as part of a system specification).

Processes are run repeatedly at scheduled times or in response to certain events. The task list of a process is divided into two parts. The first part, or initialization, is executed only once: the first time the process is run. Whenever the process is triggered thereafter, only the rest of the task list is executed.

Processes come in two kinds: sporadic and periodic. A sporadic process is to be triggered at unpredictable times, as may be required by some random event. A periodic process is to be triggered at regular intervals according to a specified period.

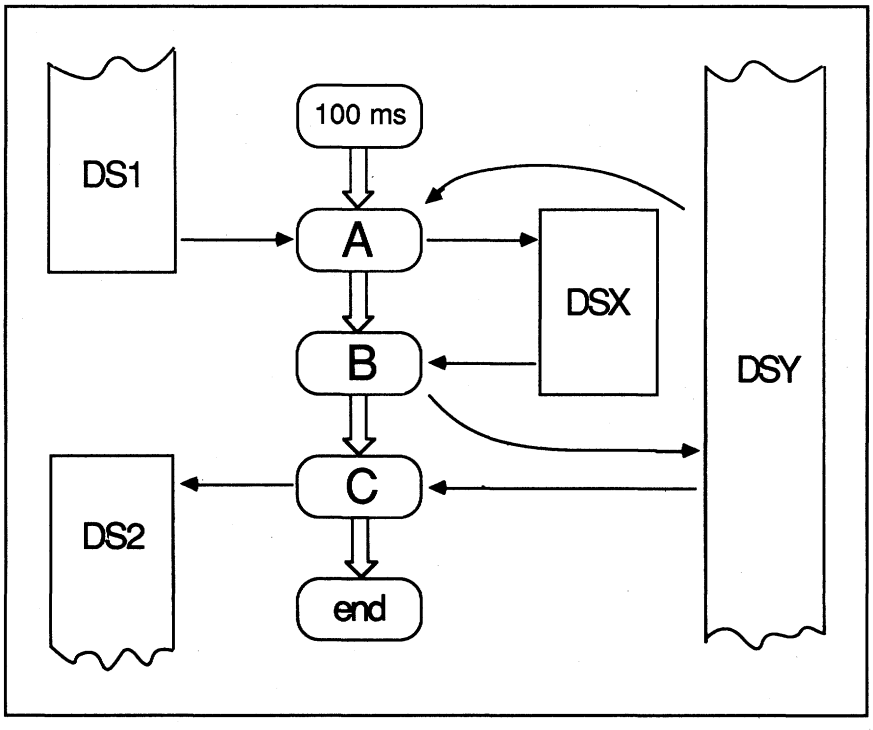



**Figure 3. A process with three tasks.**

Figure 3 illustrates a process to be triggered every 100 ms. It consists of three tasks, appearing in the task list in the order *ABC*. Task *A* reads from the global data set DS1 and from the local data set DS*Y*, and writes to the local data set DS*X*. Task *B* reads from DS*X* and writes to DS*Y*. Thus *A* and *B* update DS*Y* using information from DS1. Task *C* then reads from DS*Y* and writes to the new global data set DS2.

The data in DS1 presumably originated in another process, or from an operating system I/O handler, and are no longer needed after they are read by *A*. The data *C* puts into DS2 presumably are consumed by another process or by an operating system I/O driver. All the data sets are long-term, except for DS*X*.

## The executive

The execution of a process is under the control of the executive, which has several functions:

*Scheduling.* The scheduler determines when processes are triggered. When a process is triggered, it becomes eligible for allocation of resources.

*Resource allocation.* The resource allocator establishes and maintains a prioritized list of ready processes for use by the dispatcher. It also manages memory for a process's code, scratch space, and data sets. In multiprocessor systems with shared memory, it also must identify the set of processors that may execute a process. In systems with bulk backing storage, memory management involves swapping modules between memory and backing storage.

*Dispatching.* The dispatcher assigns processors to tasks of ready processes. Each processor executes its own copy of the dispatcher, using one set of agendas in shared memory. When the dispatcher reaches the end of an agenda's task list, or when the process is aborted, the dispatcher terminates the process.

Figure 4 illustrates how these three components of the executive work together and how the agendas provide information used by each component.

## The scheduler

The scheduler is responsible for triggering processes. It is invoked whenever an event that may trigger a process occurs. By consulting programmable tables, it determines which processes should be triggered and passes them on to the resource allocator.

Each process has an enable/disable flag. The scheduler does not trigger processes that are disabled. This allows simple adaptation to changes such as for recovery from faults.

Sporadic processes are triggered in response to asynchronous events such as hardware interrupts or software signals passed on by the dispatcher as a direct effect of the execution of a scheduling task in another process.

Periodic processes are triggered in response to events that originate as interrupts from hardware timers or clocks, and represent the passage of time. The scheduler ensures these interrupts arrive at the required times by controlling the hardware devices that generate them and by interpreting them when they arrive.

Each periodic process has a specified period (the time interval in milliseconds) until it is triggered. The period is used in one of two ways, depending on whether it is firm or stretchable. If it is firm, we have

next trigger time =
  last trigger time + period

(for example, the process is to be triggered each second). If the period is stretchable, we have

next trigger time =
  last start of execution + period

(for example, there is to be one second between runs of the process). With a stretchable period, the actual period between runs will increase with heavy loading as the delay between triggering of a process and the actual start of execution increases.

## The resource allocator

Whenever the scheduler triggers a process, it passes the process's identification code to the resource allocator. The resource allocator then readies the corresponding process for execution. This involves two functions: memory management and process allocation.

The memory management function loads all required modules (procedures and long-term data sets used by the tasks within the process) not currently resident. It also reserves a block of memory for the process's workspace.

As Rex is now implemented — this is not a property of the Rex virtual machine — the memory manager monitors how each current process is using its modules and applies a load/unload strategy optimized to reduce disk activity and response time. The physical disk storage is organized to support this strategy.

Certain processes may be defined to have all required modules permanently resident. Other individual modules may be defined to be permanently resident.

The process allocation function takes

over once a process's modules are all available in memory. Process allocation maps the new process onto the set of processors authorized to work on it. Each process can be constrained to any subset of the physical processors in the hardware configuration.

Allocation to a set of processors means each processor can perform any task within a process. Process allocation thus can be used to limit interprocessor contention and allows, for example, a processor to be dedicated to only fast-response processes.

Process allocation concludes by advising dispatchers that the process is ready to be executed. The resource allocator does this by computing the dispatching importance for the process and inserting it into the ordered ready list, which the dispatcher scans to select processes. The ready list only changes when the resource allocator inserts a ready process or when a process completes execution and is removed from the list.

All required resources remain allocated until the process terminates. The resources are released, and then become available for reuse.

In Rex, workspaces are recovered immediately, but the actual recovery of memory occupied by code and long-term data sets is postponed until it is needed to satisfy further requests. Thus, if a process is triggered again before the storage occupied by modules it uses has been reassigned, the process will not need to fetch those modules back. Rex incorporates strategies to reduce needless swapping based on module size, usage rate, and whether a module needs to be copied back to backing storage.

Better time performance can be achieved by prefetching only the required modules at the process level rather than by using demand-paging-based virtual memory schemes. This policy is based on the assumption that processes are not large and that minimizing delay is paramount.

In systems in which all code and data sets are permanently resident, swapping is not needed, and memory management is reduced to assigning workspaces. If there is sufficient memory, workspaces may be allocated statically at system generation. All intertask linkages then become static, improving efficiency. If there is only a single processor, the resource allocator need only enter the process in the ready list.

## The dispatcher

The dispatcher is invoked whenever a processor completes a task. It chooses the next task for the processor to execute. It does this by scanning the ready list until it finds a process the processor is authorized for and whose next task is not blocked. The order of processes in the ready list governs the dispatcher.

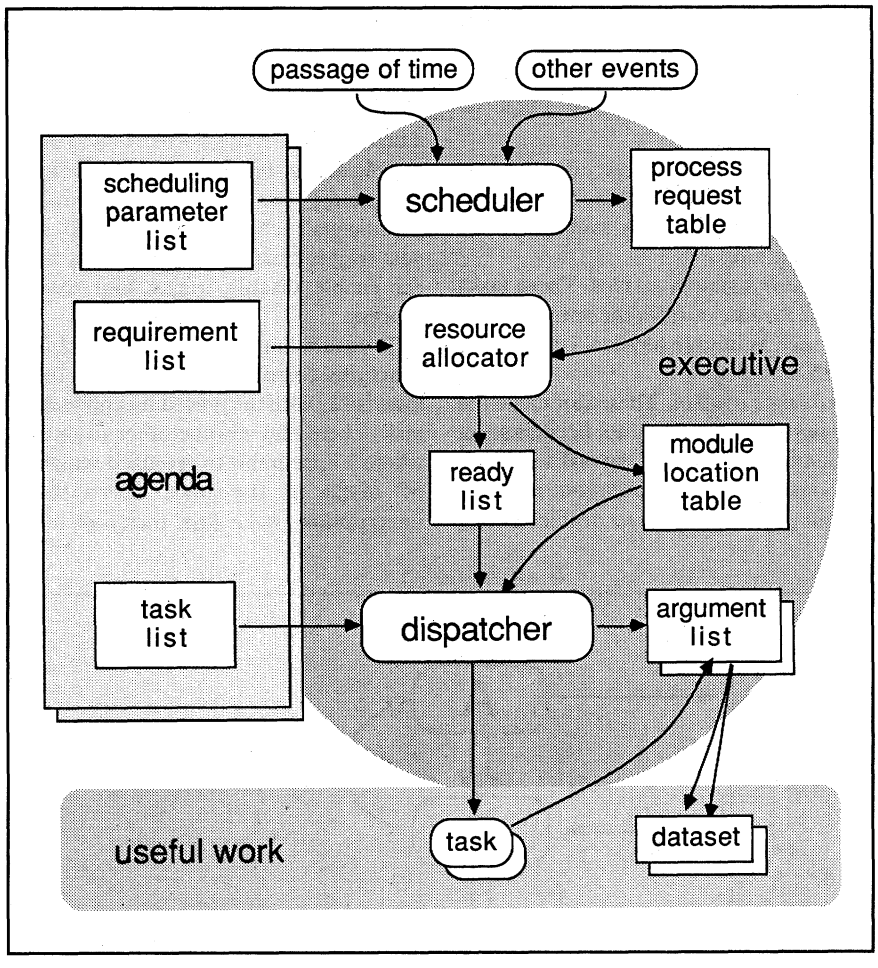



**Figure 4. Components of the executive.**

**Priority.** As mentioned above, the resource manager determines the order of processes in the ready list. It does this by computing a measure of dispatching importance for each process and keeping the ready list ordered by this ranking.

The dispatching importance of a Rex process is a function of two parameters. One is the priority of the task; the other is the length of delay. That is,

delay + priority =
dispatching importance.

Delay in dispatching a process is the difference between the current time and the time it was triggered. Since the current time is the same for all processes, and

delay = current time − trigger time,

the dispatching importance of a process is given by

priority − trigger time =
dispatching importance.

One virtue of this scheme is its adaptability. If all processes have equal priority, dispatching reverts to first-in, first-out. If grossly different priority values are assigned, it approximates fixed priority dispatching but ensures against total starvation of low-priority processes. This lets a system ride out short-term overloads.

This scheme has worked well in practice. The architecture readily permits other schemes, however. Moreover, because the calculation of dispatching importance is only done when a process is triggered, more complex schemes can be adopted without increasing the dispatcher overhead.

**Executive tasks.** In addition to normal application tasks, which call for invocations of procedures, a task list may contain special predefined tasks that control the

executive's action. These tasks are interpreted immediately by the dispatcher rather than being dispatched as an application. Rex supports several kinds of executive tasks:

*Synchronization.* These tasks impose constraints on the dispatching order within a single task list for a multiprocessor configuration.

*Operating system service.* These tasks can call for service from the operating system (such as initiate physical I/O on port *n* with buffer *x*).

*Sequence control.* These can make the dispatcher deviate from its normal strict sequential execution of the task list. Such tasks may use the condition code returned by the preceding task to decide among alternate control paths. This allows branching and looping to be implemented within a task list. A special sequence-control task skips the initialization part of a task list after it has been executed once.

*Scheduling.* These tasks can interrogate or change the values of scheduling parameters such as priority, period, and next trigger time. Other scheduling tasks can enable, disable, trigger, and abort a process.

To ensure that a sequence-control task functions correctly, one must execute it indivisibly with the preceding application task. There may also be other situations where it is desirable to execute a sequence of tasks in the task list without interruption. Each task therefore has a concatenation bit. If this bit is on for a given task, the dispatcher must dispatch the next task in the task list (on the same processor) as soon as the given task is complete — without preempting the processor.

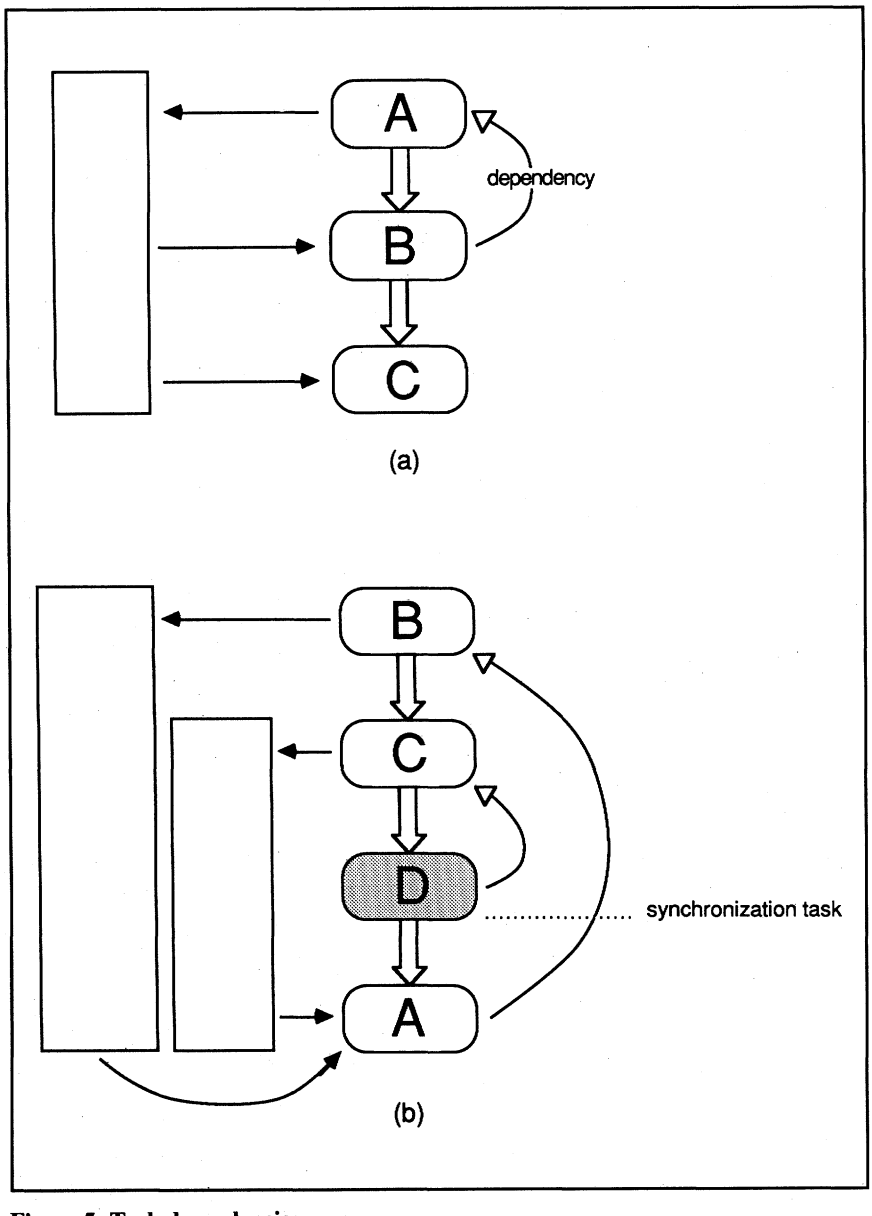



**Figure 5. Task dependencies.**

**Mutual exclusion.** When tasks sharing access to data might be executed concurrently by different processors, the dispatcher must provide mutual exclusion between them.

If communication is between tasks within a single process, mutual exclusion may be enforced by static sequencing constraints expressed in the task list. Such constraints can be used to ensure that, if a task accesses a data set written to by a previous task, it is not started while that previous task is being executed (by a different processor). Sequencing constraints are inserted in the task list automatically by a system assembler using intertask dataflow analysis based on the dispositions of parameters.

Such constraints are enforced by a dependency code and an in-progress flag attached to each task. If the dependency code is nonzero for a particular task, it is interpreted as the offset within the list of another (previous) task that must not be running when the current task is started. Since the dispatcher assigns processors to tasks sequentially within each task list, execution of further tasks for the process is suspended until the indicated task completes.

Figure 5 illustrates the use of dependency codes. In the example shown in Figure 5a, task *A* shares data sets with tasks *B* and *C* (but *B* and *C* do not share data sets). We can enforce *B* and *C* waiting for the completion of *A* by putting *ABC* in the task list, and setting the dependency code of *B* to indicate that it must wait for *A*. The dispatcher will start *A*, but will be forced by the dependency code of *B* to wait until *A* has completed before it starts *B*. Since the dispatcher will not consider *C* until *B* has been started, *C* is also forced to wait.

Conversely, if we want *A* to be executed only after both *B* and *C* have completed, we can create a dummy task *D*, put *BCDA* in the task list, set the dependency code of *D* to make it wait for *C*, and set the dependency code of *A* to make it wait for *B*.This is illustrated in Figure 5b. The dispatcher will not start *D* until *C* completes, and *A* is not considered until *D* is started. Since *D* is trivial, the scheduler tries to start *A* once *C* completes. The dependency code of

*A* refers to *B*, so *A* must in turn wait for *B* to complete. This permits *B* and *C* to be executed concurrently on different processors, but requires both to have finished before *A* can start.

In the above example, *D* is an executive task — more precisely, a synchronization task. Where such tasks are required, they may be inserted automatically by the system assembler. An alternative is to eliminate the need for such synchronization tasks by using a more general task list data structure that permits specifying dependency of a task on more than one preceding task. Rex does not now support this for reasons of simplicity.

If communication is between processes that share memory, or communication is to the I/O portion of the operating system, data set locks may be used to provide synchronization. The dispatcher locks a data set by linking it to a task and unlocking it when the task completes. The dispatcher also ensures locks are honored.

If the communication is between processes in separate computers not sharing memory but connected via a channel, the communication is treated as if it were through two separate data sets. The operating system serves as an intermediary.

It seems reasonable to expect that this form of communication could be made transparent through the introduction of implicit synchronization and transfers between remote copies of a single virtual data set. No attempt has yet been made to implement such transparent distribution of data sets with Rex.

## The operating system

The operating system hosts the executive on a particular hardware configuration. It serves as the first shell surrounding the hardware core. There are three primary groupings of operating system functions: system services, configuration management, and fault processing.

**System services.** The operating system provides a real-time clock and utilities that manage physical I/O. The I/O utilities provide services in two forms. Interrupt handlers, which are invisible to the application program, bring input data into buffers and transfer output data from buffers. Utility procedures can be invoked by application processes to initiate I/O operations and to wait for I/O operations to complete.

Operating system services, such as initiating an I/O operation, are requested by processes via operating system service tasks. These tasks are similar to the synchronization tasks used to control concurrency between tasks within a single process in that they could conceivably be supplied automatically by a high-level system assembler.

Data communication between the operating system and the application system is via data sets. Such data sets may implement different forms of buffering on a case-by-case basis.

The real-time clock and any other operating system events can trigger the scheduler and call for an application process to be run.

Hardware interrupts are always handled first by operating system interrupt handlers, whose primary responsibility is to ensure that data is not lost. If such an interrupt calls for action by an application process, the operating system signals the scheduler to trigger the process. The process will begin execution as soon as its priority permits, subject to the limitation that any tasks currently being executed must not be preempted.

***Hardware interrupts are always handled first by operating system interrupt handlers, whose primary responsibility is to ensure that data is not lost.***

Thus, while operating system interrupt handlers may preempt a processor from a task, this preemption is transparent to the task. (That is, interrupt handlers do not write on any data sets accessed directly by an executing task, and their execution times are short enough that they can be adequately accounted for as a bounded percentage overhead on the processor.) The processor is always allowed to complete a task before any other task can be dispatched on the same processor.

The operating system also includes the bulk storage controller, which interfaces the resource allocator with the backing storage subsystem.

**Configuration management.** Configuration management reconfigures the hardware architecture. Modern mainframes can include spare processing and memory components. The Rex approach to real-time control facilitates such reconfiguration because each physical processor has its own dispatcher — disabling a physical processor is equivalent to disabling the corresponding dispatcher; other dispatchers can take up the load. Also, the memory management portion of the resource allocator can adapt to a dynamically shrinking memory resource. Configuration management would cooperate with the memory manager to remap the remaining memory into contiguous space.

Configuration management also deals with the networking between mainframes. Redundant channels and retransmission protocols are automatically managed by the operating system.

**Fault processing.** Fault tolerance is an important aspect of many real-time computing systems. The fault processing portion of the operating system includes self-testing routines as well as other checks to detect failures. Once a fault is detected, error recovery routines determine the appropriate response. This can involve reconfiguration, triggering high-priority applications recovery processes, or restarting the parent process from some previous checkpoint.

Rex can be logically extended to incorporate checkpointing. We are considering analyzing the effects of communicating corrupt information to automatically determine where checkpoints should be inserted into each process's task list to permit recovery of lost data. Static recovery tables may be a practical additional product of the high-level system assembler required to produce the agenda.

The integration of fault processing and configuration management features with the Rex executive has not yet been implemented and tested. This is, however, one of the areas of current research where the Rex architecture seems to offer particular promise.

The system architecture described here has a number of virtues we hope to exploit in future research, as outlined below.

The architecture offers great potential for reuse of application software components. In fact, it allows for use of these components as user-defined, high-level instructions.

The use of such high-level instructions is discussed here only as an assembly-level process, and only two levels of abstraction (the components and their assembly into a task list) are allowed. The designer is thus constrained to think in terms of rather simple building blocks, which may not be the appropriate level of abstraction for his problem. Research into more powerful system specification languages, for which the term system compiler would be appropriate, is needed.

Because a system's global structure and scheduling characteristics are separated from the details of the implementation and execution of the components, the prototyping, development, validation, and maintenance of an application can be viewed more simply and more systematically. This offers potential for automatic generation of software from high-level specifications.

Particular virtues to be exploited here are the guarantee that concurrent tasks will not interfere with one another and that different implementations (for example, multiprocessor versus uniprocessor) use the same application programs without change.

The architecture supports application systems requiring backing storage with automatic management of swapping. In fact, resource requirements and their management are overlain onto the application system rather than built into it. Potential extensions in fault tolerance and adaptive hardware and software reconfiguration areas are promising.

Also of significant promise are extensions to include host architectures with multiple heterogeneous general- and special-purpose processors, all orchestrated by a single high-level system program. There are no inherent limitations in the architecture that preclude such an approach.

Finally, adaptations of the architecture to accommodate standard programming languages such as Ada seem promising. One interesting possibility is the automatic translation of the native structures of such a language into procedure-like chunks that can be managed by an executive. This technique could be used to impose more precise management of time-critical computations than the standard language itself provides.

Another possibility is the automatic generation (or selection from a library) of manageable chunks, coded in a language like Ada, to meet a specification expressed in a higher level language. We are exploring this path, both to exploit the standard executive architecture and to gain more understanding of the future role of Ada in real-time applications. □

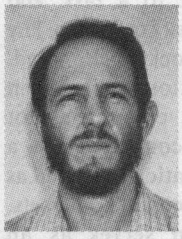


## Acknowledgments


The Washington Technology Center, the University of Washington, and Boeing Aerospace supported the writing of this article. The authors are also indebted to the referees for their constructive comments.


## References


1. J.G.P. Barnes, "Real-Time Languages for Process Control," *Computer Journal*, Vol. 15, No. 1, 1972, pp. 15-17.
2. N. Wirth, "Toward a Discipline of Real-Time Programming," *Comm. ACM*, Vol. 20, No. 8, Aug. 1977, pp. 577-583.
3. H.A. Schutz, "On the Design of a Language for Programming Real-Time Concurrent Processes," *IEEE Trans. Software Eng.*, Vol. SE-5, No. 3, May 1979, pp.248-255.
4. D.M. Berry et al., "Language Constructs for Real-Time Distributed Systems," *Computer Language*, Vol. 7, 1982, pp. 11-20.
5. *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, US Department of Defense, Washington, DC, Jan. 1983.
6. D.R. Cheriton et al., "Thoth, a Portable Real-Time Operating System," *Comm. ACM*, Vol. 22, No. 2, Feb. 1979, pp. 105-115.
7. *Introduction to the iRMX 86 Operating System*, Intel Corp., 1982.
8. *VRTX/1750 User's Guide*, Doc. 591613, Hunter Ready, 1985.
9. G. Scallon, "Description of the Functionally Oriented System Simulation (FOSS) for Software/Hardware Design," Boeing Aerospace Report D180-18617-1, Seattle.


**Theodore P. Baker** is a professor in the Computer Science Department at Florida State University. While coauthoring this article, he was on sabbatical leave at the University of Washington, where he studied tool design for real-time software development.

Baker received his BA in 1970 and PhD in 1974 in computer science from Cornell University.

His address is Department of Computer Science, Florida State University, Tallahassee, FL 32306-4019.



**Gregory M. Scallon** is an industrial research fellow with the Washington Technology Center, working on real-time software development methodologies. At Boeing Aerospace, he is applying advanced software research to a Strategic Defense Initiative program.

Previously, he worked for ITT and Texas Instruments. Scallon received a BSEE from Seattle University in 1965.

Scallon's address is Boeing Aerospace, PO Box 3999, Seattle, WA 98124.