LENGTH: STRING intrinsic that returns the dynamic data length of a STRING variable.

MARK: Used to mark the current top of the heap in dynamic data memory allocation.

MEMAVAIL: Returns number of words between heap and stack in data memory.

MOVELEFT: Low-level intrinsic for moving mass amounts of bytes.

MOVERIGHT: Low-level intrinsic for moving mass amounts of bytes.

POS: STRING intrinsic returning the position of a pattern in a string variable.

REWRITE: Procedure for opening a new file.

RESET: Procedure for opening an existing file.

RELEASE: Intrinsic used to release memory occupied by variables dynamically allocated in the heap.

SEEK: Used for random accessing of records within a file.

SIZEOF: Function returning the number of bytes allocated to a variable.

STR: Procedure to convert long integer to string.

IMM Service

Routines: Set of over 50 procedures that provide access to I/O and management services.

An example of a segmented program is the program development system itself. The root segment of the program development system simply displays a choice menu on the program development system terminal and then accepts a selection from the keyboard. This results in a segment procedure to be loaded from disk that implements the choice (i.e., editor, filer, etc.)

A disadvantage of segmented programs is that its loaded code cannot be shared among a number of users, as the instantaneous state of each user's code space is not constant but depends upon the particular state of each program's segment overlay. For example, as program development system is an overlayed program, every program development system user who logs-on causes a new copy of the program development system code to be loaded into memory. If, on the other hand, a number of identical applications use a non-segmented program, then only one copy of the program code would actually exist in each general purpose processor **942**, **944**. The individual virtual machines created for each instance of the application would map the virtual machine code space to the single copy of code and share it.

A more rational way to organize large multiuser applications is to design a system using a number of cooperating, memory-resident processes. This brings two advantages:

(1) Better response due to no segment-loading overhead.

(2) Reduction in real memory space taken with multiple users due to code-sharing of the nonsegmented components of the application system.

I. Pascal Linker

The Pascal linker allows precompiled procedures to be linked to form an object code file that can be loaded and run in an ideal machine. A reference to an external procedure or function can be made within a separately compiled module, provided that the referenced procedure has been declared as external. The linker attempts to resolve references between a set of modules submitted to it, or by selecting named precompiled procedures from within a library of code modules, if one is supplied. It is also possible to specifically replace all calls to a procedure internal to a module with a procedure supplied at link-time. This allows a new or test version of a procedure or function to be globally replaced without recompiling each module.

A link stage is required for programs that run outside the program development system environment and which use the ideal machine monitor service routines for the following reasons. First, most of the extensions to standard Pascal are implemented as functions and procedures called from the user program. Some of these calls are satisfied intrinsically by the ideal machine interpreter—an example of this is any of the character string functions. When compiled, a call to one of these functions generates a single P-code instructions (with an argument) that performs the requested function in one P-code instruction execution cycle. These functions are effectively micro-coded into the ideal machine's instruction set.

However, other extensions call internal program development system procedures implemented as P-code routines. When a program is being run under the program development system (by typing EXECUTE from the program development system terminal) many of the extensions used in a program are serviced by the program development system itself. Examples of this are the GET & PUT intrinsics—using these standard intrinsics actually generates calls to routines in the program development system that drive lower level I/O functions. Such a user program is referred to as a Level **4** process with the program development system providing a Level **3** operating environment which divorces the Level **4** application program from the underlying architecture of the ideal machine and system **100**. This is the environment that the editor operates in, and a user would normally test subsections of his application by using the program development system execute function.

Eventually, however, an application program operates in a stand alone environment, and is initiated not from a terminal command under program development system, but by the JSAM. Therefore, any procedural routines normally supplied by the program development system to implement intrinsics and extensions (i.e., GET and PUT) must be linked with the application before it can operate independent of the program development system.

The second requirement for a link stage relates to the low-level ideal machine monitor service routines. Once again, a subset of these calls trap directly to micro-coded routines in the ideal machine monitor—typically those that call REX functions directly (for example, Send a Packet). Other routines require a P-code "front-end" to map a higher level call into a specific set of primitive ideal machine monitor calls—for example, one ideal machine monitor service routine reads the memo in an indexed dataset via a sequence of basic ideal machine monitor intrinsic calls. These P-code drivers must be linked into the users application.

Note that these procedures, when linked to the user porgram, reduce by a small amount the code-space available for the application, as they reside in the same 64K-byte address space available in a single ideal machine. However, the amount of space taken by extension, intrinsic and ideal machine monitor service routines is not great—typically less than 2K-bytes.

When a program is running under the program development system after an EXECUTE command, the application code is co-resident with program development

system in the same virtual machine, and hence the amount of space available for the application is less than 64 K-bytes (program development system takes approximately 16K-bytes). Normally, the various segments and procedures of an application are tested in this environment.

J. Program Testing

The program development system, as an interactive single-user system, does not provide any specific testing tools for SPM programs. As the program development system resides in an ideal machine which executes P-codes, SPM code cannot be checked out in the program development system environment itself. To test SPM code, two facilities are available:

(1) Monitor functions from a system programmer's console for in-system testing

(2) Use of an auxiliary extension board for in-system and stand-alone testing.

For Pascal programs, a high degree of static checking is carried out by the Pascal compiler, which not only checks for syntactic and semantic errors, but also provides extensive type-checking of variables across assignments, procedure calls, etc. A successfully compiled Pascal program will mostly contain only logical errors at run-time. As these programs are, by the nature of the Pascal language, of a modular design, individual sections can be tested by executing them under the program development system with run-time errors reported to the program development system console. For execution tracing, the module can be seeded with READs and WRITEs which will interact with the program development system terminal as the program runs. The Hex Dump utility can also be used to print diagnostic lists of data files generated by the program as an alternative source of trace information. This latter method is then available for producing dumps of trace files generated by the complete application system running outside the supervision of program development system. The interactive nature of the editor and compiler ensures a fast error correction cycle to eliminate logical operation problems.

Privileged programmers use the program development system to generate SPM programs to be run in the host space of the system **100**. SPM programs are necessary when a new device handler, a hardware-specific transient, a time-critical transient or a time-critical job supervisor is required.

Authorized personnel using a system programmer's terminal **270** can load and run programs for test purposes, interacting with the processor environment in which the program is loaded. From a system programmer's terminal **270**, the program, scratchpad and data memory of any processor can be displayed and altered. Packets can be sent to, and received from, the process using the test program, and dump tables can be specified in the event of the process trapping the processor. As a system programmer also has access to all of the functions available to a system operator, the system log can be used to collect run-time trace packets sent from the test program, and the verified program can then be installed in a program library. All of these functions can occur during normal system operation.

K. Ideal Machine Implementation

The implementation of the ideal machine software in a general purpose processor is presented here for interest only. Except for any desired application program, no user programming is required in a general purpose processor.

The ideal machine monitor subsystem is a collection of SPM processes resident in a general purpose processors high-speed program memory. Two main processes (i.e., processes with an allocated scratchpad context, and a system process ID (SPID) of 0) can be identified: ideal machine monitor subprocess driver (IMMS) and general purpose processor resource manager.

All requests for ideal machines are sent as create process requests to IMMS by JSAM, specifying a P-code Program ID. Ideal machine monitors creates a subprocess within its own main context for each such request. The shared program code for this subprocess provides two functions:

(1) ideal machine monitor interpreter driver (IMMI), and

(2) ideal machine interpreter (IMI).

The IMMI function of the subprocess generates a request to the general purpose processor resource manager to load the program specified by the request into data memory and return to IMMI the data and code segment register contents needed to map out the loaded code in memory. If the P-code is already loaded, the existing copy is pointed to. IMMI then initiates running of an ideal machine by starting the ideal machine interpreter (IMI) decoding and executing the P-codes contained in the code space now pointed to by the mapping registers.

IMMI provides intrinsic services on behalf of IMI during process execution, including interfacing to REX functions. IMMI also deallocates resources used by the ideal machine when the P-code process terminates by calling the general purpose processor resource manager.

IMMI and IMI functions are provided by a single wubprocess and use the suspend option when waiting for the completion of REX functions (I/O, Event Management, etc.)

Every P-code process running in its ideal machine employs an IMMI/IMI subprocess to implement the basic machine emulation. However, due to SPM code sharing, the only SPM program-code required to be resident in the general purpose processor program memory are single copies of ideal machine monitors, IMMI and IMI. The general purpose processor resource manager is a P-code process running in system ideal machine.

Major kernel system processes implemented in Pascal use a main process invocation of IMMI/IMI. As each main process has its own scratchpad context, it can be assigned a unique system bus ID (SBID) by JSAM. This is necessary for those Level **2** system processes (for example, SYSDEV) requiring an instant global address (SBID) to which any process can send a packet. (When an IMMI/IMI process runs as a subprocess under ideal machine monitors, the subprocess identity cannot be pre-established, thus such subprocesses cannot be assigned global addresses.) In certain cases, SBIDs are also assigned dynamically to permit use of the packet rerouting facilities provided by the inter processor communications system to implement fault tolerant dual processing schemes.

Finally, this approach provides a system ideal machine with its own scratchpad context, reducing the process context-switching overhead to a minimum. In contrast, a normal ideal machine operating as a subprocess must save its operating context in a shadow context in data memory, to permit sharing of the ideal machine monitors main context.

RESIDENT EXECUTIVE (REX)

A. Overview

The resident executive (REX) is a firmware subsystem integral to each system **100** processor. It provides the resources management which software processes require to effectively use the hardware facilities of the standard processor module **500** (SPM), common to all processors. Software processes executing in a processor perform under the management of REX, which can support the operation of multiple, concurrent software processes. REX's responsibilities include the following service areas:

(1) Logical initialization of the standard processor module **500** on power-up.

(2) Loading of programs into the standard processor module **500** on request.

(3) Allocation of memory resources to processes.

(4) Scheduling and dispatching of processes.

(5) Process suspension and event management.

(6) Input/output and packet transfer services.

(7) Inter-processor utilities and subroutine calls.

(8) Timer utilities.

(9) List processing utilities.

(10) Interrupt handling.

(11) Diagnostics.

In the system **100** each process contains an identical coph of REX, combined with a processor-specific extension to REX (called generically, EXREX). EXREX manages the particular hardware extension of the standard processor module **500** which, together with the standard processor module **500** form one of the five different system **100** processor types.

Communications between REX and an active resident process is via subroutine calls. Communications between a specific processor's REX and a process in another processor is vis packets transmitted over the inter-process communications (IPC) network. There is a close relationship between the multiple copies of REX and the kernel system management process, JSAM (job scheduling, allocation and monitoring subsystem manager). Each REX is periodically requested by JSAM to report the status of its resources and this information is used to allocate required resources to the dynamically changing workload imposed on the system.

An understanding of REX services is important to users of ideal machines for applications programmed in Pascal, although such applications programs do not interface with REX directly. Ideal machines created in general purpose processors **942**, **944** (GAPs) provide a logically separated and hardware fence protected environment for Level **3** application systems. Nonetheless, most of the image machine monitor services routines called via Pascal program procedures make direct and obvious use of the underlying REX callable subroutines of the host general purpose processor.

A system **100** consists of a collection of independent, tightly-coupled processors, each with its own central processing unit, program, scratchpad, and data memories. Although working cooperatively with other processes in the system, each within itself is an autonomous processing unit. At a system-wide level, each provides a useable resource to which work, in the form of software processes, is allocated. At the processor level, this allocated workload together with its hardware must be managed by the processor itself, free of any higher-level system-wide organizational structure.

The resident executive provides this management. Each processor, of the five different types that can exist in a Delta **2** system, has up to 32K-words of program memory, 12K of which is read-only memory used to house a copy of the REX program code. Each contains an identical copy of the basic REX code and has its own personal extension to REX that manages the hardware resources specific to that processor type. This extension is referred to as EXREX and is housed in an additional 4K-words of read-only memory.

Because REX provides a set of fixed, well-defined user functions and is in integral part of the basic hardware of a standard processor module **500**, it is considered to be as much a primitive component of a standard processor module **500** as are the instruction set, memories, registers, and basic logic of the board itself. Thus the standard processor module **500** processor is a self-managed hardware system providing a reliable environment in which to host multiple concurrent processes, all contending for the use of the processor's fixed resources. A user process executes compiled basic machine instructions to achieve its specific goal but calls on a REX function to communication with other processes, acquire additional machine resources, use timers, or perform other general activities. REX management not only provides the applications programmer freedom from concern with many hardware-specific operations, but greatly increases the overall reliability of the system since most of the more complex activities needed by the user, and normally callable by basic machine instructions, are provided by mature, well-tried REX routines. Additionally, it reduces the problem of resource and activity contention by concurrent processes in a single processor by allowing for a structured, disciplined way to handle allocation of processor resources to processes. Thus the low-level program visibility of a processor presented to a programmer is both the standard processor module **500** instruction set and the set of REX-callable functions.

The life-cycle of a process within the system **100** is used as an example of the kinds of services REX provides within each processor.

A process is the basic unit of work from which system-wide jobs are built. To execute, a process needs a processor with the available resources necessary to host the associated program (memory, attached channels, execution time, etc.). Processes are initiated by a component of the kernel system called the job scheduling. Allocation, and monitoring subsystem manager (JSAM), which bases its decision as to which processor should host a process by comparing the quantity and types of resources required with those available in the system's processors. JSAM maintains current resource inventories by a regular background dialogue with the REX of each live processor in the system. (Process requests can be sent to JSAM by any active process in any processor in the system.)

Having chosen a suitable host for the requested process, JSAM sends to the REX of that processor a request to initiate the process. If the program code needed by the process is not already in program memory, REX loads the code froma disk-resident program library and creates the process. Once running, the process acquires from REX whatever resources are necessary to perform its task.

REX can initiate and manage a large number of processes, suspending those awaiting resources or an event and scheduling the execution of those currently dis-

patchable. When a process is running, it can make subroutine calls to REX to request a wide range of services including:

(1) Dynamic allocation of scratchpad and data memory.

(2) Management of and access to user-defined lists allowing a process to efficiently maintain a variety of single and multi-thread queues and stack structures.

(3) Communications with other processors in either the same or different processors.

(4) Management of any number of general purpose timers on behalf of the process.

(5) Semaphore services for user-defined resource sharing and activity synchronization.

(6) Management of hardware interrupts.

(7) A set of general purpose I/O services allowing data transfer to be performed between and among devices, datasets, and processes owned by the requesting process, regardless of where it is located in the system.

(8) A large number of general purpose utility and computational routines designed to ease the burden of user-programming of standard processor module processes.

During process activation, REX manages the interface between the process and any outside events related to it (for example, the receipt of packets destined for it). A process can be suspended, and thus become dormant for a number of reasons: waiting for a response to a communication, a timer to fire, or a resource to become free. When a process is suspended, REX enqueues the process until the event occurs, and dispatches whichever process has become the next-most-eligible to use the machine.

When a process terminates, REX returns all interprocessor resources used back to the general pool, and notifies JSAM of termination.

REX also manages hardware interrupts and provides a utility routine for each possible external interrupt condition. Many interrupts relate to the detection of hardware errors within the processor. REX manages the trapped error-condition allowaing inspection from a system terminal and diagnostic processing.

When a processor is first powered-up, REX takes the hardware through a predefined series of initialization and diagnostic routines. After successful completion, REX reports to JSAM that the host processor is available for work allocation.

B. Memory Management

Scratchpad management provides all resident programs in a standard processor module **500** including REX, with an orderly way of using the scratchpad memory resource. Order is preserved by requiring that all allocations and decallocations be accomplished using REX's scratchpad routines exclusively.

Each processor contains 4096 16-bit words of high-speed scratchpad memory, organized as 32 pages of 128 words each. Each page is arranged as eight packets of 16 words each (the structure of a scratchpad and interprocessor packet are obviously similar and logically equivalent).

The first page of scratchpad memory, page zero, is reserved for common use. Information which must be inaccessible to all resident processes in the host processor, such as the date and time, is maintained here. REX uses page one as its own private scratchpad area or context. The remaining 30 pages are available for allocation by REX either for user-process contexts or dynamic working storage.

A context is an area of scratchpad memory allocated by REX at the time each process is created. The number of scratchpad packets allocated is specified to REX in the create process request. This is generally kept to a minimum since context packets are allocated for the life of the process and provide "registers" for system control information as well as a static working area of storage used by the process as it desires.

The initial context allocation to a process may not meet all of the processes' lifetime requirements for scratchpad memory. This may occur because the size of the context is physically limited, or because it is wasteful for a process to request long-term allocation for waht are often short-term needs. REX satisfies such requirements for dynamic working storage by allocating and deallocating packets to each process on demand. Such requests are made on an individual packet basis. The amount of scratchpad allocated can thus expand and contract as necessary to meet the changing demands of the process during its lifetime.

Since a context must be allocated to each main process in a processor, the number of main processes is limited to the number of contexts. To keep this number as large as possible, REX allocates dynamically from those pages from which a context has already been allocated and maintains all spare packets available from previously allocated contexts on the available packet queue. Each time a context is allocated, packets not requested from the context's eight packets are placed on this queue. Because REX prefers to have as many full-page contexts as possible, each time a context is deallocated, spare packets from the same page are removed from the available packet queue. If the packets are contiguous with the deallocated context, they extent it. If not, they are placed on a free packet stack to await the return of the rest of the packets in the page. The free packet stack is composed entirely of packets which were once on the available packet queue, but have since been removed because the context from the page of which they were a part is no longer allocated. The free packet stack is maintained as a single-linked stack, using REX's standard scratchpad list facilities. When the remaining context packets are returned, they are all removed from the free packet list and used to extend the unused context back into a full page. REX will not allocate from the free packet stack unless the available packet list is empty. In this way, REX reconsolidates complete context pages.

The page consolidation function is performed by REXIDLE (REX's permanent background process which continually scans the available packet queue.

Besides managing scratchpad memory as a resource, REX also provides a set of functions for maintaining lists of individual packets on single-linked stacks or queues for user-defined operations.

Data memory management provides all resident processes in a processor, including REX, with an orderly way of using the data memory resource.

Each processor contains 64K 22-Bit words of data memory. Each word consists of 16 bits and six error correcting code (ECC) bits. The ECC bits permit detection of all 2-bit errors and correction of all 1-bit errors.

Initially, a minimal work area of 4K-words is assigned to REX. All remaining available memory is assigned to extension board REX (EXREX), the needs of which vary depending on processor type. If, during initialization, EXREX determines that this assignment

is excessive the surplus memory blocks are transferred back to REX.

Whatever the size of the area initially assigned to REX, it may at times exhaust its supply of data memory. EXREX can aid REX by providing temporary extension blocks on these occasions. During physical initialization, EXREX informs REX of the addresses of its allocation and deallocation routines, its data memory area, and the maximum length of the extension blocks it can provide. Then whenever REX requires additional data memory, it requests a block from EXREX and later, after the need for a block passes, it is returned to EXREX.

The data memory management routines used by REX and available to all processes implement a buddy system of storage management. Blocks of any length up to the maximum may be requested by a process. However, available data memory is maintained only in blocks of 4, 8, 16, up to 32,768 words, that is, the lengths of all available blocks are in powers of two and in the range 2^2 to 2^{15} inclusive. A request for a block of any other length is satisfied by allocation from the next-larger available block, successively smaller power-of-two blocks until the request is satisfied with a total allocation which exceeds the requested size of the least possible amount. Since the smallest available block is four words in length, up to three extra words may be allocated to satisfy a request. The unused memory remaining after request allocation is retained as a number of smaller available blocks.

During allocation, it is possible that an available next-larger block does not exist from which to allocate the request. In this case, a still larger available block can be successively split into two "buddies" of equal size until a block of the next-larger size is obtained. Since the length of each available block is a power of two, the length of each buddy thus obtained is also a power of two.

When a previously assigned block id deallocated, unless its original length was a power of two, the block cannot be made directly available. Instead, smaller blocks, the lengths of which are each successively larger powers of two, are split off from the deallocated block until the length of the remainder is also a power of two. It can then be made available.

Each time a new available block is created, either by deallocation or as a result of splitting, all other available blocks of the same length are examined to determine if the buddy of that block is also available. Since the origin in data memory of each block is a multiple of its length, the address of the buddy may be easily determined. If the buddy is available, both blocks are recombined to form a new available block of the next-larger size. Recombinatio is performed by REX's background process, REXIDLE.

Program memory in each processor consists of at least three blocks of 4096 16-bit word ROM and up to 65,536 16-bit word RAM for REX and EXREX. Both ROM and RAM are 50 nanosecond memories. Since only RAM can be loaded dynamically, one might expect REX to consider only RAM as a manageable resource and to accept ROM as a non-alterable preassigned resource. This is not the case, however. REX treats all program memory, both RAM and ROM, as a manageable resource and wherever possible does not differentiate between the two. This uniformity of treatment presents a number of important advantages, particularly in the area of program loading.

A request to load a program into program memory can be accepted regardless of the memory type of the target area. Programs cannot actuqlly be loaded into ROM, but part of the loading operation is to verify that the expected checksum which accompanies the program being "loaded" matches the actual checksum computed from the data in memory. Thus a program on disk can be "loaded" into ROM to verify that the two are identical.

Another advantage of treating memory uniformly is that, as long as loads are requested for all programs, RAM and ROM can be interchanged without altering any code. If the target area is RAM, the area written to is changed. If the target area is ROM, it is not. This flexibility greatly facilitates testing and debugging.

REX allocates blocks of program memory in lengths of up to 4K-words. Blocks can be requested by length alone, or by both address and length. Each block of available program memory (whether RAM or ROM) is described by a program memory element (PME), an internal data structure maintained in a queue by REX in its own data memory area.

When a block is requested by length, REX searches the program memory queue for a block of sufficient size from which to satisfy the request. The search is begun at the head of the queue, and the first block of sufficient size encountered is selected.

If a block is selected from which to make the allocation, the program memory element for the block is changed (the length and address so that it then describes only that portion of the original block which remains after the allocation is made). If no residual exists, the program memory element is instead deleted from the queue and returned to available data memory. On the other hand, if the allocation cannot be made because a block of sufficient size does not exist, the request is rejected and an error return to the calling process is made.

When a user requests that a previously allocated block be returned to available program memory, an attempt is made to recombine the returned block with any existing unassigned blocks on either side of it in the program memory address space.

A program can consist of from one to nine relocatable load modules, each limited to a maximum length of 4K-words, and up to 32 overlay modules. Load modules can be either two types: primary or secondary. Primary are program-related, that is, only one re-entrant copy need exist in memory no matter how many processes share its use. Secondary are process-related and a separate copy of each must exist for each active process. Overlay modules are loaded by explicit request and may overlay primary or secondary modules. (Generally, overlay modules do not exist in re-entrant programs). A program must have at least one primary module loaded into program memory before a process can be created.

Programs are loaded by REX in response to create process request packets. Such requests assume the implied condition that the program be loaded only if it is not already present in program memory. Because the transfer of a program from disk is a relatively time-consuming operation, REX creates a subprocess to perform each load.

Initially, the subprocess sends SYSLOAD (part of the system directory subsystem resident in a disk data processor **934, 936** a packet requesting the transfer. SYSLOAD (via the SYSDIR program load process

PGMLOAD) responds with a packet containing information describing the program, including the quantities types and memory required. The loading subprocess allocates that memory and creates all control blocks necessary for each of the program's load modules.

The subprocess then requests that the program be loaded. As the program packets arrive, they are put into memory one by one until the entire program has been loaded. The subprocess proceeds to calculate a checksum from the loaded program, and compares it with another passed to it in the last packet received from PGMLOAD. The subprocess accepts the load if an error has not occured during the transfer, and the two checksums match. Otherwise, the load is rejected (if rejected, the load is not retried). In either case, the load subprocess then terminates.

Once loading of a given program is begun, any further program, load requests to the same process are enqueued to allow the one in progress to complete. Eventually, after the load completes, the subprocess initially created to perform the load for the first request proceeds to process each of the other requests, in the order in which they were received.

It may occur that a program load is required but a contiguous area of available program memory is not sufficiently large enough to accept it. In this event, REX re-examines each program in the program load list, considering those not currently being used by a process. If the required program can be accommodated by overlaying an unused adjacent one, the request is accepted; otherwise, it is rejected. Note that unused programs are not automatically deleted to avoid having to reload if they are required again in the future.

C. Process Management

Work performed by a processor other than handling interrupts is accomplished by the execution of processes. A "process" is a program and associated set of dynamic data (that is, its context) provided execution time by REX.

For each process, a control data structure referred to as a process control block (PCB) is allocated in data memory describing the attributes and status of that process. REX maintains several different lists of process control blocks which allow control of the execution phases a process can be in. The process control blocks listed are organized in descending order of dispatching priority, the dispatching priority determining which of the members is most eligible for attention. One list consists of all process control blocks for dispatchable processes while the remaining lists are made up of those for dormant or nondispatchable processes; for example, processes awaiting for an event to occur before they can proceed.

A process can create subordinate processes that execute using the same scratchpad context as their creator, and these can each create still other processes using that same context. In this way, a total of up to 256 processes can be created to execute using one scratchpad context. The first process created in a given context is called the "main" process. All other processes spawned by the main process are called "subprocesses".

A single process, executing alone in its own context, is referred to as a "simple" process. A main process and a set of subprocesses, all of which share a common context, are referred to as a "compound process".

When a main process is created, it can be defined as a "bypass" process designated to receive not only packets addressed to it, but those addressed to any of its subprocesses whether the subprocess actually exists or not.

Any process (main or sub) to which JSAM allocates resources can create another main process, and any main process thus created can itself request the creation of still other main processes. For certain kernel system subsystems, the original request to JSAM causes a multiprocess block of resources to be allocated. In such cases, the request can be directed to the same processor in which the requesting process itself is executing.

When REX receives a create process request packet (from the requestor, generally JSAM), the program load list is first examined to determine if the required program already resides in program memory. If it does not, the program is requested from the system library and loaded. The created process is then immediately available for execution.

A process can request the creation of subprocesses, and any subprocess thus created can itself request the creation of still other subprocesses (up to a maximum of 255). Each uses the same scratchpad context as the creating process, possibly with a "shadow context" overlay kept in data memory. The program to be executed by a subprocess must have been loaded with or by its main process and must reside in program memory at the time the subprocess is created. A subprocess is created by a direct function call to REX.

Process deletion refers to the elimination of one process in response to a request from another, enabling a process working as a control node to order the termination of one of its subordinates. JSAM, however, also uses this function to order the termination of an orphaned process (one whose control node has failed).

It is important to distinguish between process deletion and process abortion. Process deletion is the execution of a planned sequence of operations by one process to elicit its own orderly demise at the request of another. Process abortion, however, is the premature removal of a process by REX as a result of an unrecoverable error detected and attributable to that process. In order to be deleted, a process must have defined a deletion entry point to REX. A subprocess can be deleted only by the process that created it, or by its main process.

No subprocess can outlive the process that created it, and an attempt to terminate a main proces with any surviving subprocesses results in a trap. For this reason, a compound process, if it is to be terminated, must delete all of its created subprocesses prior to terminating.

Each process if assigned a bi-level dispatching priority consisting of a process class and a rank within that class.

Five process classes are defined, number from zero to four. Processes assigned to class four have the highest priority; those assigned to class zero have the lowest (processes with class zero are reserved for use by REXIDLE). Within each of the four classes, processes are ranked from zero to 255, the highest rank being 255. The kinds of processes normally assigned to each class are listed as follows:

Class 4: Interrupt-initiated background processes, imperative timers.

Class 3: Normal events.

Class 2: Delayed timers (e.g., timeouts), lot-priority processes, time-initiated long processes.

Class 1: All other processes (except REXIDLE).

Class 0: REXIDLE.

The initial dispatching priority can be assigned when the associated program is assembled, or the assignment can be deferred until the process is actually loaded. In either case, after a process begins execution, various REX routines allow the process to dynamically alter its own priority or, subject to certain restrictions, that of another process.

REX selects the highest-priority dispatchable process for execution, that is, the one whose process control block is at the head of the dispatchable queue. Once given control, a process is allowed to execute until it voluntarily relinquishes control or is interrupted by the occurrence of a hardware interrupt.

A process can relinquish control voluntarily or involuntarily. It voluntarily relinquishes control in one of three ways.

(1) By calling a wait routine to wait for one or more events to occur, including calling an input or output services routine in a mode which invokes suspension until the requested service completes.

(2) By calling a routine to signal processes' desire to terminate its owh execution.

(3) By calling a resource management routine, in a mode which invokes a wait if the resource is not immediately available. (This is considered a voluntary relinquishment of control since the process chooses to call this routine.)

The first method of voluntarily relinquishing control, calling a wait routine, is used whenever a process cannot proceed until some external event occurs. The event might be the completion of an I/O operation, the expiration of a specified time interval, the receipt of an unsolicited packet, or any other condition defined for the process.

Calling a wait routine causes the calling process to be made nondispatchable until a specified or an unsolicited event occurs. Control is returned to the process when it again becomes the highest-priority dispatchable process.

The second method is used when a process has no work left to perform. This may be because it has successfully accompoished all of its assigned tasks, or because some insurmountable problem has forced it to abandon its mission prematurely.

Once a process terminates, control is never returned. Therefore a process must complete all internal processing prior to termination, including the return of any owned resources and termination of all subprocesses.

Calling a resource management routine, the third method of voluntarily relinquishing control, need not result in a loss of control. However, if it cannot allocate the resource because it is not available, the process does lose control. When the resource becomes available, the process will again be made dispatchable.

A process involuntarily relinquishes control whenever a hardware interrupt occurs. After the interrupt is serviced, control may or may not be immediately returned to the interrupted process.

As previously explained, all processes are assigned to one of five process classes. Processes in one class are allowed to interrupt those in a lower class. Processes in the same class, however, are not allowed to interrupt others in that class, regardless of rank.

When a process of one class interrupts that of a lower class, the interrupted process is said to be "suspended". Since five process classes are defined and processes cannot interrupt others in the same class, at most four processes, one in each class except the highest can be suspended at any one time.

The servicing of a hardware interrupt can result in making dispatchable a process whose class is higher than the interrupted process. In this case, the interrupted process is suspended and the higher-class process given execution control. Other interrupts, after servicing, can result in execution control returned to the interrupted process.

A dormant or suspended process can again become dispatchable in one of three ways:

(1) An event for which it is waiting occurs.

(2) The process is restarted.

(3) A previously unavailable resource becomes available.

When a process voluntarily reliquishes control by calling a wait routine, it normally must wait for its process control block to come to the head of the dispatchable queue, and then for the active process to relinquish control before it is again allowed to execute. (It can, however, be restarted by an interrupt handler or another process.) When a process waits for a specified or an unsolicited event to occur, its process control block makes its way back to the head of the dispatchable queue in one of several ways, depending on when the event occurs and how it was initially defined.

If, at the time the wait routine is called, the event has already occurred, the process remains dispatchable but is process control block is replaced on the dispatchable queue in descending order by dispatching priority. The process must then wait for all processes ahead of it to execute before it is given another turn.

If the wait routine is entered before the event has occurred, the process is made nondispatchable and its process control block is moved to the event quque. When the event occurs, the process control block is returned to the dispatchable queue in the usual priority order to await its next turn at execution.

A nondispatchable process can again be made dispatchable by a call to a restart routine. This call can be made by an interrupt handler or another process.

When an interrupt handler restarts a process, the restarted process, if it is assigned to a higher priority class than is the active process, forces the suspension of the active process and immediately becomes the active process.

When a process restarts another process, the restarted process is made dispatchable but the currently active process is never suspended in its favor. The restarted process must wait its normal turn to execute.

A request to restart a process is acceptable only if the process is waiting for an unsolicited event to occur. If the process is nondispatchable for any other reason, the request is ignored.

A process made nondispatchable due to the lack of a requested resource is made dispatchable when the resource again becomes available. The process regains control at the same point and with the same register contents, as if the resource had been available initially. The temporary loss of control is thus transparent to that process.

Most processes perform some initialization prior to beginning data processing. They may need to set up tables, create subprocesses, or complete various other functions that generally prepare the process to handle events. To ensure that this initialization is allowed to take place, each process is created dispatchable. The

process will, therefore, execute at least once and perform any required initialization at that time.

If a process is to accept any unsolicited packets the function codes contained in those packets must be defined to REX. A process accomplishes this by allocating and initializing a function code branch table. Function code branch table is created in a 16-word area of data memory, with one word for each possible function code (the first word of the area corresponds to function code zero, the last to function 15). Each word of the area must be initialized to contain either the address where control is to be passed if a packet with the corresponding function code is received, or zero to indicate that packets received with the corresponding function code are invalid and are to be discarded.

It should be noted that a process can change the function code branch table at any time, and the process can temporarily enable or disable one or more function codes.

The structure of a compound process, where each member process uses the same scratchpad context, greatly facilitates communication among the components. However, conflict for the use of the limited scratchpad space may also be introduced.

To avoid this problem, a "shadow context" whereby eight packets selected from the scratchpad context can be saved is allocated in data memory for each process of a compound process. Then each time a compound process is allocated to execute, REX verifies that its content in scratchpad reflects the content of the shadow context of the executing component process. If necessary, REX swaps into scratchpad the executing process shadow context, saving the overlayed context packages of the suspended process in its shadow context.

A shadow context is not automatically defined for a main process when it is created but if a creating process defines a shadow context for itself, then identical ones are automatically defined for all subprocesses in its compound process.

If a main process requires a shadow context or a subprocess needs one different than that of its creating process, the required shadow context can be defined and allocated data memory.

D. Event Management

An "event" is any change of state (either "hardware" or "software") that can be recognized and communicated to a process.

Hardware events generate interrupt requests for 15 possible interrupt conditions in each of three levels. For each level, a separate interrupt branch vector contains the addresses of all interrupt handler routines designated to service the interrupt conditions assigned to that level. Control is transferred directly to the designated interrupt handler routine whenever an interrupt request is generated at a higher level than is currently being serviced.

Software events do not generate interrupt requests, nor are they limited to a small number of predefined conditions. Rather they are defined dynamically by processes, and their number is limited only by the amount of available data memory. Each of the following conditions, when properly described to REX, constitute a software event for a process:

(1) The receipt of an input packet.
(2) The expiration of a time interval.
(3) The completion of an I/O operation.
(4) The termination of a subprocess.
(5) The receipt of a signal from another process.

The term "event" when used in the following paragraphs refers exclusively to a software event.

For each event, REX maintains an event control block (ECB) in data memory. The event control block describes the event to REX, and contains information used by REX to recognize the event and control execution of the process associated with it.

Generally an event control block is created in anticipation of an event, that is, before the event actually occurs. Events for which event control blocks are created in advance are referred to as "solicited".

In some cases, however, an event is not certain to occur. In other instances, even though it is sure to occur, an unpredictable amount of time may pass in waiting. In the interest of conserving data memory, event control block creation can be deferred until after the event occurs. An event of this type, for which an event control block is not created until after the event occurs, is referred to as "unsolicited".

Events can have time limits associated with them, referred to as "timed" events. Those that do not are called "untimed".

A timed event is one which must occur within a specified interval of time. A packet transfer, for example, is typically a timed event. If the transfer does not complete within a given time interval, a problem has definitely occurred.

The event control block which describes a timed event contains an expiration time specified when the event is defined. If the event occurs prior to this, it is said to have completed "normally" and if it does not, it is said to have completed "with error". The error is called a "timeout". If an event "times out" and then occurs, it is no longer defined and is ignored.

An untimed event is one which need not complete within any specified interval of time. The event control block which describes an untimed event does not contain an expiration time, and therefore the event cannot "time out".

Interval timers are, for the most part, treated in a manner identical to timed events. The event for an interval timer is the expiration of the specified time interval. However, timeout in this case is not considered an error.

When an event occurs, the event control block describing that event is "posted" to the process associated with the event. Whether or not the process is then made dispatchable to service the event depends on the status of the process itself, and the return-point addresses specified. If the process is waiting for the event to occur and the supplied return-point is nonzero, the process is made dispatchable from the return-point address. An executing process must explicitly check its list of posted event control blocks to determine if an event, either unsolicited or solicited (identified by a reference number), has occurred.

Note that even if a suspended process is made dispatchable by the occurrence of the event and is of the same class but of a higher rank than the currently active process, control remains with the active process until it voluntarily relinquishes control.

When an event occurs, the event control block describing that event is "posted" to the process with an event completion code. This code indicates whether the event completed normally or with error and if completed with error, the code also indicates which error.

An event completion code is made up of an error group and code. Codes belonging to group zero indi-

cate normal completions and thos belonging to all other groups indicate error completions.

Whether the event completed normally or with error determines at what point the associated process will be dispatched to process the event. If the event completed normally, the process is dispatched at the primary return-point address. If the event completed with error, the process is dispatched at a secondary return-point address.

A 16-word function code branch table is defined for each process, each one-word entry in the table corresponding to one of 16 function codes associated with each packet or signal that a process will recognize. The entry contains the address where control is to be passed when a packet or signal with a given function code is received.

For each process, a function code mask determines whether events with a given function code are solicited or unsolicited. Each bit in the mask corresponds to a particular code and if a bit is set, then a packet or signal with the corresponding function code is unsolicited.

E. Input/Output Functions

The high degree of processor specialization in the system **100** is particularly evident in the area of I/O control. A number of different special purpose processors may participate in performing this function for the many attached I/O devices.

Each input, output and control channel for each device in the system **100** is attached to a processor designed specifically to host handlers for that class of device. Disk drives are attached to disk data processors **934, 936**; magnetic tape drives, terminals, printers, and data channels are attached to interactive services executives **702, 706**, telephone, operator terminal voice channels, and real-time data channels are attached to real-time executives **406, 408**.

In each processor, a system process known as a device handler is hosted for each type of device channel attached to that processor. The responsibility for all communication with and control of a device channel rests with its assigned device handler. A user-process application in the system **100** does not communicate directly with I/O devices. Instead it makes requests to device handlers that interact with the device to perform the requested functions.

Generally the device handler does not reside in the same processor as the user-process. Device I/O for a user, therefore, involves inter-processor communication in the form of packet interchanges over the main buses.

Because communication with all device and data set handlers is as such, it is beneficial to both user and handler designer to provide a standard framework within which these packet interchanges can be defined. The REX I/O service routines (IOSRs) make up this framework.

The input/output services routines (IOSRs) are a set of subroutines, each affecting a prescribed sequence of packet interchanges with a device or data set handler. Each packet interchange performs some basic I/O function and together provide a complete interface between the user-process and device or data set handler.

Generally such an interchange proceeds as follows: a user-process request an I/O function be performed by issuing a subroutine call to the appropriate input/output services routine, which formats the user's request into a request packet and forwards it to the specified device or dataset handler. The handler then either performs the requested function responding with a confirmation packet to the input/output services routine, or rejects the request responding with an error packet. In either case, the input/output services routine extracts any pertinent information from the response packet and returns it to the user-process. The manner in which a specific handler responds to a given function depends on the nature of the device, and the particular handler implementation.

The input/output services routines allow the user to perform I/O operations in either two modes: "wait" or "no-wait".

When a user issues a request in wait mode, the input-/output services routine places the user in a wait state suspending him until either the requested operation completes or some other previously defined event occurs (the choice is the user's). A process exclusively using wait I/O cannot, therefore, overlap operations and it never has more than one I/O operation outstanding at any one time.

When a user issues a request in no-wait mode, the input/output services routine returns control to the user once the requested operation is initiated which is as soon as the request packet is sent. A user operating in this mode can have several overlapping operations outstanding at the same time.

The input/output services routines monitor I/O operations using two reference values supplied by the user when requesting an input/output services routine operation. The first, a channel reference number in the range of one through 255, is uniquely associated with a device or dataset. This number is assigned when the device or dataset is opened, and released when the device or dataset is closed. The second, an access reference number also in the range one through 255, is uniquely associated with a particular input or output operation for a device or dataset. This number is assigned when the I/O operation is initiated, and released when the operation is completed.

Once a device or dataset has been opened, all accesses must specify the assigned channel reference number to select a particular device or dataset. After an access to an individual logical data record has been issued, all data transfers to or from that record must specify the assigned access reference number associated with that record.

The reference numbers assigned by the user when an operation is initiated are returned when the operation is completed. The user can, therefore, determine which of several outstanding concurrent input/output services routine operations has completed.

A time limit, expressed by the user, can be applied to all I/O operations. The user can specify a time interval (in seconds) which when added to a predefined system timeout value determines the actual timeout interval for the operation. The time begins when the request packet leaves the outstack.

One parameter in each input/out services routine is the address of a parameter list in data memory. The list is constructed by the user prior to calling an input/output services routine, and contains a set of parameters specific to each input/output services routine call. The contents of the list are not changed by the input/output services routine.

F. Time Management

The variety of time management facilities provided by REX are divided into two categories: those related

to interval timing, and those concerned with date and time-of-day maintenance.

All interval timing functions within a processor are controlled by that processor via one 16-bit hardware register in the processor's standard processor module CPU 504. The register is called the programmed interval timer (PIT).

To maintain internal consistency, REX alone controls the programmed interval timer, satifying the various timing requirements of individual processes by creating a "virtual programmed interval timer" for each timing interval required and manipulating the hardware programmed interval timer to operate all of the virtual programmed interval timers. A process can explicitly or implicitly request that an interval timer be created.

An explicit request is made whenever the process calls one of the REX time management routines to request that the elapse of a specified time interval be defined as an event.

An implicit request is made whenever the process calls a REX routine (other than a time management routine) to define an event which must occur within a specified period of time.

Regardless of whether the interval timer is created explicitly or implicitly, it is placed on a single list along with all other event control blocks associated with timed events. REX maintains the event control blocks on this "timer" list in chronological order by expiration time.

The expiration time at the head of the list is used to determine the current value to be loaded into the programmed interval timer. When an event control block reaches the head of the list, the difference between its expiration time and the current programmed interval timer contents is calculated, and the programmed interval inter is reloaded with this value. When the interval expires, the event control block is removed from the list and posted to its process.

JSAM, a system process, periodically provides REX with the system data and time in both binary and BCD formats. REX maintains the most recently received values in scratchpad page zero. JSAM sends updates to successive processors approximately every 50 ms. In a system with a full complement of 32 processors, each processor receives a new date and time about every 1.6 seconds.

For some applications, this discrepancy from true system time is not acceptable. REX provides a correction factor in page zero equal to the number of programmed interval timer ticks since the last update was received. The user can use this value to adjust the date or time-of-day to obtain a more accurate result.

REX performs the functions described previously by setting and resetting the programmed interval timer. It operates as follows:

By loading the programmed interval timer with the appropriate count, REX selects a time interval requested by a process. Each 256 instruction cycles, the programmed interval timer is decremented by one (that is, "ticks" occur every 34,133 microseconds). The 16-bit programmed interval timer can be decremented a maximum of 65,536 times and thus define a maximum time interval of 2.237 seconds.

When the programmed interval timer is decremented past zero (that is, when the specified interval expires), a Level 1 interrupt request is generated.

The programmed interval timer continues to be decremented, however, and must be reset under REX control within 1024 ticks (approximately 34,953 milliseconds), or a Level 3 watchdog timer interrupt is generated to signal an overrun.

Programmed interval timer is considered about to "tick" if it will decrement within the next 16 instruction cycles, and it can be tested for this condition. If a tick is imminent, it should be allowed to occur prior to programmed interval timer readout to avoid errors in calculations by use of a noncurrent value.

G. User-Defined Resource Management

All processes acquire, use, and release resources. These might include memory space, datasets, I/O devices or could be slightly more abstract such as execution time or permission to access a list. In fact, almost anything of interest to a process can be considered a resource for that process.

Some resources are of interest to more than one process and must be shared. Access to shared resources must be controlled to prevent the activities of one process from interfering with those of another. This control can be vested in a formal resource manager. The REX data memory manager, for example, controls all accesses to REX's pool of data memory.

Alternately, the processes involved can exercise the resource control necessary by cooperating with each other. This shared resource control is implemented using flags to synchronize the activities of the cooperating processes. REX supports two kinds of flats: binary and general. These, and the routines for using them are described and followed by a discussion of how such flags can be used to control access to shared resources.

A binary flag is a flag which has two possible values: true or false. REX represent such a flag as a resource control block (RCB). Each resource control block has a unique name by which it and the flag it represents are known. The value of a binary flag is true if a resource control block with the same name exists and the value is false if one does not.

A general flag is a flag which can take on any integer value in the range zero to 32,767. It is also represented by REX as a resource control block, and has a unique name by which it and the flag it represents are known. The current value of a general flag is stored in the resource control block (the value of a binary flag is stored in the resource control block, but is always one).

A resource can be successfully shares as long as all processes which access that resource synchronize their activities to avoid conflict. The manner in which this synchronization is completed depends on the relationship which exists among the processes sharing the resource. Three relationships are common: (1) competition for a single resource, (2) competition for a pool resource, and (3) production consumption.

These relationships, and methods for using flags to synchronize process activity to avoid conflict, are examined in more detail in the following paragraphs.

Processes which compete for a single resource must synchronize their activities to avoid conflict with one another. For example, processes updating the same word in memory cannot do so simultaneously and still maintain the integrity of the contents. Rather they must synchronize their activities so that each is allowed to complete any outstanding update operation before a new one is begun by another. In such cases, synchronization can be accomplished using a binary flag. Access to the resource can be denied or permitted depending on whether the value of the flag is true or false.

In operation, the first process to find the flat set false (no resource control block) in its attempt to access a single resource is permitted to acquire it. A resource control block is now created so that other processes attempting access to the same single resource find the flat set true and are suspended. After the using process releases the resource, the next process (if there is one) enqueued for that resource is given access to it. In this way, the resource is passed from one process to the next, with each in turn granted exclusive access to the resource for as long as is needed.

Processes may compete for a resource from a pool of undifferentiated resources, that is, any one of a number of identical resources may satisfy the requirements of any one of a larger number of processes, all of which require one or more such resources. The available resources must, therefore, be shared. Without a formal resource manager, the participating processes must synchronize their activities to avoid conflict with one another.

For example, suppose a number of processes all share a smaller number of identical magnetic tape units. Access to this pool of resource units can be controlled using a general flag as a resource counter. Each time a unit is.allocated, the count of available unit (value of the flag) is decremented by one; each time a unit is deallocated, the count is incremented by one. The count, therefore, always equals the number of units available for allocation. If a process attempts to use a resource when the count is zero, it is suspended until a resource unit (a magnetic tape unit in this example) is released by another user.

For a given resource, two or more processes may relate to each other as producer(s)-comsumer(s), that is, one or more processes may produce a resource that one or more processes may consume. Where such a relationship exists, participating processes must synchronize their activities to ensure proper resource production consumption, and avoid conflict between competing producer(s) and consumer(s).

For example, suppose one process (the producer) fields request packets and adds them to a list. Several . over processes (the consumers) remove these request packets, one at a time, and respond to them. This flow of request packets from producer to consumer is maintained using a general flag as a resource counter. (In this example, the actual list of request packets would itself be a shared resource. Access to it could be controlled using a binary flag). Each request packet is added to the list, the count (value of the flag) is incremented by one; each time a request packet is removed, the count is decremented by one. The count, therefore, always equals the number of request packets available for processing. Any consumer attempting to remove a packet from the list when the resource count in the general flag is zero is suspended until the producer adds another packet to the list. If the producer adds packets to the list faster than they are removed, the list lengthens. If the consumers exhaust the supply, they must wait but are restarted as soon as more request packets become available.

A general flag can be simulated using a binary flag to manage access to a shared counter. Similarly, a binary flag can be simulated using a general flag with a count of one. What then, are the advantages and disadvantages of each?.

Binary flags are simpler to use since only two routines need be called and fewer instructions are required. However, if there is no contention for a resource, a resource control block will be created and deleted each time the resource is accessed. This takes time and also means that a process could be denied the resource, not because it is not available but rather because the resource control block cannot be allocated for lack of memory.

With general flags, the creation and deletion resource control blocks is explicit. It is possible to create a resource control block once, then access the resource several times before deleting the resource control block. However, using general flags requires more instructions since four separate routines need be called.

In summary, if a process accesses a particular resource infrequently, a binary flag is appropriate and should be used. If the resource is accessed often, it may be better to use a general flat (with a count of one) so that the resource control block is created and deleted only once for all accesses

H. List Processing

Many applications utilize lists as a means of maintaining order in a group of structurally related data elements. Performing operations on these lists is commonly referred to as list processing

The maintenance of any list, regardless of structure, requires that certain common functions be performed (for example, the addition or deletion of an element). REX supports these functions for user lists maintained in data memory

A variety of list structures is also supported by REX. A list has a corresponding list control block and may be either a stack or queue, single linked (forward step pointer) or double-linked (forward step pointer and backward step pointer) All lists supported by REX are linear.

The list control block for a single linked stack has three words defining a list type code (C7-C$ =1,0, 0,0), a top pointer and a forward step pointer A double-linked tack control block includes list type code (C7-C4 _1,0,0,1), a top pointer, a forward step and a back-pointer. The single linked queue control block ward step pointer, has a list type code (C7-C4-1,1,0,0), a head a tail pointer and a forward step pointer The double-linked queue control block includes a list type code (C7-C4=1,1,0,1), a head pointer, a tail pointer, a forward step pointer, and a backward step pointer and thus requires five words of storage.

Elements of a resident list in data memory are represented in one of two formats: "standard" or "primitive". The standard format is intended to meet most demands; however, when space considerations are foremost, the overhead required by this format may prohibit its use. In this case, it may be necessary to use the primitive format, though this requires that some capabilities be sacrificed.

The standard format provides broad flexibility in the handling of lists. It also permits multi-threading of list elements to any predetermined level and movement of list elements from list to list without regard to differences in individual list structure. Each list element represented in this format consists of a control and data section.

The control section is made up of a single two-word control header, followed by one or more four-word control segments. One control segment is required for each list on which the element is to appear at any one time. For example, a triple threaded list element would require a 14-word control section, that is, a two-word

control header followed by three 4-word control segments.

The data section can be contiguous with or separated from the control section, or may be omitted entirely. Access to the data section is made via a pointer (first word) in the control header.

The second word of the header includes a first field defining the list length and a second field defining the number of segments in the control section. The four words of each segment include a pointer to the first word of the header, a link control block (LCB) pointer, a forward link pointer and a backward link pointer.

The primitive format is used where list element lengths must be kept to a minimum. It provides a compact means of representing single-threaded list elements, that is, those which appear on only one list at a time.

Each list element in this format consists of a two- or three-word control segment equivalent to the first two (or three) words of a control segment in a standard element. These words define a link control block (LCB) pointer, forward link pointer and optional backward link pointer. The data section of a primitive link element (if one exists) is known only to the user, not to the list processing routines themselves and follows immediately the two- or three-word control section.

I. Inter-Processor Communications

A packet is the unit of information transferred between processors on the main buses of the system **100**. It consists of 16 words, the first of which is a header used by an executive services processor **916, 918** to route the packet to the proper receiving processor. The packet is presented to the main bus via a processor's outstack and received from the bus into a processor's instack. The placement of packets into an outstack and removal from an instack are controlled by REX.

Within the system **100** a process is uniquely identified by a 20-bit process ID (PID) consisting of three components: a bus ID (BID) assigned when the processor enters the system on power-up; a context ID (CXID) related to the scratchpad context used by the process; and a subprocess ID 9SPID) with a zero value if the process is the context's main process. Component processes of a compound process have subprocess IDs of non-zero value. Since all processes within a given processor have the same bus ID, the context ID and subprocess ID are sufficient to identify which process within a processor a packet is address, and together form an internal process ID (IPID).

There are two basic packet types defined: process packets and immediate packets. Packets with a context ID other than zero are routed to an existing process in the receiving processor. These are referred to as process packets because they establish communication and control activity between processes in the system. The format of a process packet allows REX to accept packets on behalf of the receiving process to carry out standard sequences of control and data interchange for that process. The format is given in the interface specification of a process, which defines what packets can be meaningfully sent to a process, and what packets a process will send during its lifetime.

Immediate packets are distinguished by a context ID of zero. This indicates that they are not addressed to a specific process, but are either related to an ongoing data interchange managed by REX or to system and maintenance activities. Such packets are handled immediately at the interrupt level by REX, EXREX, or specially-defined handlers. There are several types of immediate packet, each of which has its own format and is handled differently. The function code in the packet distinguishes one type from another and determines which handler is invoked to process the packet.

The most universally used immediate packet is the immediate data packet. It is used by REX to effect all data transfers in the system in response to input/output requests. Immediate data packets are created by the outstack handler and processed as input by the appropriate receiving handler.

Force-load memory packets facilitate various system development and debugging activities by permitting memory to be loaded directly with the contents of the force-loaded packet.

JSAM bus test packets instruct REX to perform various tests related to the specified ports. These packets contain a list of headers so that as long as the tests are successful, they can be daisy-chained to each processor in the list. JSAM is notified of any bus-port test failures.

Maintenance packets are used to communicate with diagnostic programs in unusual situations. The contents of these packets are defined by their users.

Several of the function codes direct to context zero are reserved for use by EXREX. This allows handlers, unique to a normal extension board (NEB), to be invoked so that special functions can be performed for the extension board at the interrupt level.

The data area of the CPEXECUTE packet contains processor program code which is loaded into REX's own program area. After inserting a program jump back to REX at the end of the code, REX executes the "program" sent in the CPEXECUTE packet. This packet is used in systems development and troubleshooting environments.

When an incoming packet has been transferred to a processor's instack (X or Y) by an executive services processor, an X-instack-full or Y-instack-full interrupt occurs in the receiving processor and control is transferred to the instack handler. The instack handler reads the packet header to determine whether it is an immediate or process packet. If it is an immediate packet, control is transferred to the appropriate immediate packet handler. Otherwise, the instack handler will find and/or create all control blocks needed for the process packet to be passed to the destination process.

Output from a processor is placed on the main bus (X or Y) via one of two 16-word hardware outstacks. The outstack handler controls the placement of data into these stacks and sets flags to enable the executive services processor **916, 918** controlling the bus to actually effect the transfer of the packet to the destination processor. REX supports two methods of output: direct and nondirect. The direct method is used to transfer exactly one packet of data, which does not require sequence control transfer structures. However, there may be an associated event control block if the output operation is to be timed, or if the user is to be notified when the data leaves the outstack. The nondirect method is used to transfer data areas that require more than one packet. This requires a transfer control block (XCB), which is a data structure used by REX to control the sequencing of multiple packets between the sending and receiving processors. Nondirect transfer also provides additional capabilities in the form of header and buffer lists. The user calls REX routines for both methods, which enqueque the request on the bus output queue.

Duplicate copies of the same data area can be transferred to two different processes with the use of a header list. The header list contains two headers, one for each of the processes to which the data is to be sent.

Several noncontiguous data area can be transferred in a single operation with the use of a buffer list. The buffer list contains the address and length of each area to be transferred.

When an output operation fails due to a port reject, REX automatically tests the port and retries the operation on the other port. If that port is closed, it retries the same port again. Retry is possible since REX retains a copy of the data placed in each outstack in a separate "shadow" packet in scratchpad. To test the port, REX sends a self-addressed port test packet through the port, permitting REX to differentiate between problems with the intended recipient process or difficulties elsewhere on the bus. If the problems are located elsewhere, REX closes that port and notifies JSAM through the remaining port.

J. REX Processes

Along with providing a large number of callable subroutine functions to a user-process, the resident executive subsystem implements a set of processes used by REX itself to fulfill its own operations. Each processor contains the program code used by REX's processes in read-only memory and on power-up initialization, two processes are immediately created in every processor:

REXMAIN: The main REX process primarily responsible for the creation and deletion of all other processes in the same processor. It also performs program loads and manages interprocessor communications.

REXIDLE: A permanent subprocess used to absorb and report all unused machine time in a processor.

REXMAIN creates additional temporary subprocesses from time to time, but these are deleted as soon as their assigned functions have been completed. For example, in order for REXMAIN to create a process, the required program must first be loaded into memory. This is a relatively time-consuming operation. Therefore, whenever a program load is required, REXMAIN creates a temporary subprocess that is used to load the program, create the process, and send a response packet back to the requester.

REXIDLE, REXMAIN's only permanent subprocess, absorbs all otherwise unused instruction cycles in a processor. A count is kept of the number of unused idle cycles in a processor. This is reported to JSAM as a measure of the level of activity in the processor.

REX's main process REXMAIN operates out of context number 1 of each processor. As its subprocess ID is 0 (for example, it is the context's main process), then the PID of REX to which requests are sent to a given processor is:

Bus ID=LBID of processor (allocated dynamically by JSAM)

Context ID=1

Subprocessor ID=0

The main packet requests sent to REX are for the creation and deletion of processes within the host processor of the receiving REX.

REAL TIME SUBSYSTEM

A. Functional Description

The real-time subsystem **230** as shown in FIG. **4** is an integrated set of hardware and software components, tailored to provide fast circuit switching and real-time processing functions. The real-time subsystem **230** can interface to a large number of synchronous data channels, each one capable of carrying a continuous, non-interruptable real-time signal. For example, in voice applications, these channels would connect to the separate telephone room subsystem **206** to transmit PCM voice signals to and from the system **100**.

The integral executive software of the real-time subsystem provides a high-level user interface to the three classes of real-time functions:

(1) Switching, to make or break the flow of data from a source channel to one or more destination channels.

(2) Processing real-time data that flows between the system **100** and one or more external channels.

(3) Effecting the transfer of channel supervisory and control messages between the system **100** and the synchronous data channels which, in voice applications, would be the telephone room subsystem channels **232, 233**.

These functions can be controlled by any process owning the channels involved, generally a Level **3** application process running in an ideal machine.

For voice applications, the channels between the telephone room subsystem **206** and the real-time subsystem **230** are duplex paths with a nominal data rate of 64 Kbps in each direction, enough for each path to carry a two-way digitized voice connection. The single real-time subsystem **230** has the capacity to attach up to 1260 physical channels, and each system **100** is able to support up to four interconnected real-time subsystems. The allocation and use of real-time channels is managed in a fashion similar to that of any other type of data channel connected to the system (for example, printers and terminals). Each channel has a unique identity and before an application can effect a switching or processing function, the channel must be acquired from the system device manager (SYSDEV) by the application in the normal way.

To reduce the number of physical connections in voice applications that need to be made between the telephone room subsystem **206** and the real-time subsystem **230**, groups of **30** real-time 64 Kbps channels are multiplexed onto single, full-duplex 2.048 Mbps synchronous data linkes **232, 233**. These high-speed links form the actual interface to the real-time subsystem **230**, and an external equipment subsystem such as the telephone room subsystem **206** performs the multiplexing and demultiplexing of individual 64 Kbps channels onto the links. In voice applications, this subsystem **206** also performs the analogue-to-digital and digital-to-analogue conversion of channel information. Multiplexed into each high-speed link is a duplex 64 Kbps control channel used to communicate with the system **206** equipment and, in the case of the telephone room subsystem **206**, control the line group controllers **302, 306** that route each channel onto the synchronous link to the real-time subsystem **230**. It also controls the line interface boards **310–318** onto which each analogue speech path terminates.

Switching functions allow external real-time channels, attached to the real-time subsystem **230** (via the external subsystem), to be interconnected. The real-time subsystem **230** provides a "logically continuous" circuit link between groups of channels, but uses electronic time division multiplexing (TDM) to actually perform the switching rather than mechanical switches. Additionally, the real-time subsystem **230** provides 508 switched simplex links which, once set up, require no

intervention or action by the system **100**. The switching function is fully programmable by executive processors in the real-time subsystem **230**, which provide the following advanced switching functions:

(1) Dynamic allocation of bandwidth through the switch channels with higher or lower individual data rates than the nominal 64 Kbps. This is referred to as a super- and sub-commutation. A higher data rate per channel is achieved at the expense of fewer switch paths through the real-time subsystem **230**.

(2) Many-to-one channel switching, used in voice applications for conferencing a number of callers.

(3) One-to-many channel switching, used in voice applications for broadcasting messages, conferencing, and recording.

A real-time subsystem **230** is configured with its own processing resources within which real-time signal processing functions are located. Data from real-time channels attached can be input to transient processes hosted by the real-time subsystem **230**. Alternatively, a process running in the real-time subsystem **230** can generate real-time data which can be output to a real-time subsystem **230** outgoing channel.

Examples of transient signal processing in voice applications are voice record and playback. These functions permit record and replay channels to be logically connected to the file services subsystem **908** to record or playback real-time voice data using random access disks as the storage media.

Before recording, the real-time data is available for processing activities such as compression or filtering. Subsequently, the recorded data is retrieved, reprocessed as necessary, and returned to an outgoing real-time channel one or more times. These facilities are used in applications such as voice and image store and forward, image processing, message libraries, etc. During recording, a compression factor of better than two to one can be achieved, depending on the nature of the data processed.

Record and playback compression are specific examples of real-time signal processing activities performed by transient software processes hosted by the real-time subsystem **230**. The real-time processors **410, 412** can be user-programmed to perform other processing functions on real-time channel data. A single real-time processor **410, 412** can sink up to 16 channels into data memory for processing; simultaneously, another 16 independent outgoing channels can be sourced by the same real-time processor **410, 412**. Typical applications include spectral analysis, filtering, and pattern recognition. The real-time processors **410, 412** are fully integrated members of the system **100** family of processors, and can host process nodes of systemwide distributed jobs. In a user-defined transient process a real-time processor **410, 412** can be initiated and controlled by a high-level Pascal primary node in a general purpose processor. In a time-critical application, a job's primary node could reside in a real-time processor **410, 412** itself.

B Switching

Channels attached to the real-time subsystem **230** need to be interswitched in many applications, as are those related to voice communications. Automatic call distribution, telephone answering, and voice messaging are all examples of applications which use the real-time subsystem **230** for their switching capabilities. These applications, realized as Pascal-sourced Level **3** job supervisor processes, communicate with the real-time subsystem **230** via an uncomplicated procedural interface within the application program process.

Tne protocol can be divided into three stages: acquisition, control and release. In stage one, the real-time channes to be switched or controlled must first be acquired by the application process. The channel acquire request is sent to the system device manager (SYSDEV), which grants ac-ess to the channel if it is currently unowned, and at the same time informs the executive software in the real-time subsystem **230**, of the job identity of the channel's new owner. In this way, the real-time subsystem **230** can restrict the use of the channel to the owning application. Precisely which channel an application acquires depends upon the application and for what particular reason it is needed. For example, in the telephone answering support service (TASS), a job is initiated by an unanswered telephone with TASS being passed as a start-up parameter. TASS then acquires the specific channel by quoting its physical channel ID (in practice, its normal seven-digit telephone number). However, if TASS later needs to dial out to a central office, any one of the free trunks connection the system **100** to the central office will suffice. After receiving an acquire request from TASS, SYSDEV will assign one of the free trunks to TASS and pass the job identity of that specific TASS (at any given time, multiple different TASS jobs can be in execution) to a real-time processor **410, 412**. In the case where an application acquires a composite multi-channel device such as an operator station **224, 226** (a keyboard/VDU together with a voice headset), SYDEV will pass back to the application both channel identifications related to the one device: the channel ID or the data line connecting the keyboard/VDU to the interactive services subsystem **252**, and the channel ID of the voice line connecting the operator headset to the real-time subsystem **230**. This second channel ID is used later in switching commands to connect the operator's headset to other voice channels.

The second stage of the real-time subsystem/user protocol, once all of the channels needed by the application have been acquired, is the control phase. Control functions are divided into two classes: (1) Switching commands that set up interconnects among owned channels and (2) Supervisory commands that cause various signals or state conditions to be generated on the channels.

If stage three, after completing whatever functions the application requires, the channels are returned to the general pool of unowned devices via a release request to SYSDEV.

Stitching commands given to the real-time subsystem **230** either set up or tear down a network of interconnected channels.

In setting up an interconnected channel network, the application can include up to seven owned channels in a single command. The parameters accompanying the command specify how each channel is to be connected to each other, and contain sufficient detail to allow any combination of interconnections to take place between the seven duplex channels permitted in the command.

A desired network interconnection, if it is to be constructed via a single command, can be represented as a 7 by 7 matrix. If each matrix element represents one network interconnect, then the elements row depicts the channel from which the data comes and the elements column depicts the channel to which the data goes. For example, to set up a three-way conference call

between two callers on channels **3** and **5** and an operator on channel **2**, the following array would be generated by the application program:

CHO: 0 0 0 0 0 0 0
CH1: 0 0 0 0 0 0 0
CH2: 0 0 0 C 0 C 0
CH3: 0 0 C 0 0 C 0
CH4: 0 0 0 0 0 0 0
CH5: 0 0 C C 0 0 0
CH6: 0 0 0 0 0 0 0

where 0 implies no connection and C implies a connection.

Having specified the desired network, the user supplies a reference number by which this network can be identified in the future (the Circuit Net ID). The real-time subsystem circuits function implements the connections and wnen successful, acknowledges back to the application The real-time subsystem **230** rejects a circuits command if an unowned or non-existent channel is specified.

A second switching command from the application is used to disconnect, or tear down a previously set up interconnect. In this case, the application has only to pass the Circuit Net ID with the disconnect command. The real-time subsystem **230** then disconnects each connection specified in the original circuits command referenced by the Circuit Net ID. If a specific interconnection is to be modified as would occur when the operator drops out to leave a simple two-way conversation in the previous three-way Opepator/Client/Client example, the application would first tear down this entire network and then interconnect the two clients with a new circuits command. Such tear down and reconnect would be totally transparent to the clients.

An application can interact with the content and status of a real-time channel in one of two ways:

(1) The real-time subsystem **230** can be used to produce signals which can be applied to a channel. In voice applications, this is used to generate tones or tone sequences such as reorder, recording-in-progress, and dual tone multi-frequence (DTMF) digit sequences.

(2) The application can interact directly with the multiplexing and interfacing hardware outboard of the real-time subsystem **230**. In applications requiring the telephone room subsystem, the state of each voice line can be individually controlled and the line status requested.

For voice applications, interface to these supervisory commands is again via simple procedures. Three main Pascal program calls exist for these applications:

(1) Signal Generator with parameters Channel ID frequency amplitude and duration.

(2) Dialer with parameters Channel ID, digits to be dialed, DTMF or dial-pluse select.

(3) Supervisory with parameters Channel ID command (0..63), command argument (0..63).

The signal generator command produces an audio tone on the identified channel with an amplitude specified by the user and an integral frequency between 1 and 3200 Hertz. The dialer function generates a train of digits on a channel, encoded either as 50 ms DTMF tones or as make-break dial pulses. The supervisory command allows an opcode and related argument to be passed to the interface controller of any individual line termination in a telephone room subsystem **206**. Typical opcodes generate off-hook condition on lines, request line status, seize ringing lines. etc.

C. Real-Time Processing

The telephone answering support system and voice messaging system are examples of applications which process and transfer data between real-time subsystem channels and disk datasets. Since one significant use of the system **100** is for such applications, a detailed overview of its operations follows. Two transient signal processing functions are invoked as necessary to satisfy the requirements for record and playback; both are hosted by the real-time processors **410, 412** of the real-time subsystem **206**. Among the system **100** processors, real-time processors have the unique attribute of being in direct contact witn the channel switching hardware of the real-time subsystem **230, 238**. This enables the record and playback functions to interact directly with data on the real-time subsystem channels. Furthermore, specific hardware extensions in the real-time processors **410, 412** greatly increase the efficiency of the speech compression/decompression algorithms used by record and playback to reduce the disk storage requirements.

The record and playback functions are implemented as Level **1** transient processes. When an application program wishes to use either of the functions, it sends a create node request to job scheduling, allocation monitor (JSAM), specifying the function ID of either record or playback. JSAM initiates a process that performs the function in an available real-time processor **410, 412** within the subsystem. The application then communicates control requests directly to the function process.

The first stage of a record/playback operation is similar to that of a switching operation. The application sets up a channel interconnect network, as in a circuits command. However, of the seven channels that can normally be specified in a switching command, only two relate to internal real-time subsystem channels carrying real-time data into or out of the record/replay function In this way, a single command from the application can both set up a channel network and then either record from it or replay to it, with up to five external real-time subsystem channels involved in the operation

For example, the network could be used by an operator during an in-call to record a message in the telephone answering support service application.

In the array, the vertical dimension defines the source of data and the horizontal dimension defines the destination. As an example, channel **1** is assigned to the operator, channel **3** to the client, channel **5** to the operator recording process and channel **6** to the client recording process. The record network map array would be defined as:

CH0: 0 0 0 0 0 0 0
CH1: 0 0 0 C 0 C 0
CH2: 0 0 0 0 0 0 0
CH3: 0 C 0 0 0 0 C
CH4: 0 0 0 0 0 0 0
CH5: 0 0 0 0 0 0 0
CH6: 0 0 0 0 0 0 0

This array would be constructed by the application and passed as a parameter in a Pascal procedure call after both channels specified had been acquired, and the record/playback processes had been started. Recorded information may be stored in either compressed or uncompressed format. The format chosen is stored along with the data so that playback can expand the data if required.

Once the setup commands specifying the record replay networks have been given, the record/replay functions are controlled by the application process in a man-

ner similar to that of a simple tape recorder. Each process can be in one of five distinct states:

(1) Initialized (for example, functions active).

(2) Ready to be activated (for example, network setup).

(3) Recording of playing back.

(4) Pausing.

(5) Terminating record or playback.

The following control commands can be sent to the record of playback processes by the user:

(1) Start Recording/Playback—Includes information which allows access to the dataset that will sink or source the data. Indexed chain datasets are used and can be either singular for efficiency or duplicated for security.

(2) Pause Record/Playback—Ceases recording or replaying temporarily and inserts a mark in the file. (This can be used later in playback to skip back to the marked location.)

(3) Restart Recording/Playback—Restarts either at the current pause mark or at an alternative mark supplied by the user in the restart command.

(4) Mark Recording/Playback—Without halting the process, a mark is inserted into the dataset at the current position.

(5) Stop Record/Playback—Ceases recording or replaying and returns to ready state.

(6) Terminate Record/Playback—By reference to a Circuit Net ID, recording or replaying cease, all data paths are closed, and the existence of the record or playback process is terminated.

(7) Record/Playback State/Status Response—Sent to the user by Record or Playback when the process enters a new state, or when a request could not be implemented (This could be unsolicited if an end-of-message occurs during playback.)

During recording, compression can take place to reduce tne amount of disk storage required. Compression of data is achieved in two ways: (1) each PCM value representing a voice (or possibly image) sample is difference encoded (assisted by a hardware look-up table in the real-time processor **410, 412**) to give an initial 1.5:1 reduction or (2) a run-length encoding process (which removes silent parts of voice data) can further increase this initial compression ratio.

The real-time processors **410, 412** of the real-time subsystem **230** can be used to host any software functions interacting directly with data on real-time channels. The real-time processors **410, 412** are located between the executive services bus **912, 914** and the real-time subsystem inernal synchronous S and T TDM buses **414, 416**. This enables them to communicate with any other processor while being able to sink and simultaneously source up to 16 channels from the real-time processor's data memory.

As a guide to the processing capacity provided by a real-time processor **410, 412**, up to eight record and 16 playback functions can be concurrently hosted by a single real-time processor **410, 412**. The number of real-time processors in a real-time subsystem **230** is limited only by the upper limit of processors in the system **100**, and thus can never exceed 22.

Real-time processors are standard processor module-assembler programmed, and generally host Level **1** transient processes used to provide signal processing services to Level **3** Pascal applications in a general purpose processor **942, 944**. However, a time-critical Level **3** application could have its primary node hosted in a real-time processor **410, 412** if necessary.

D. External Interface To The Real-Time Subsystem

Logically, each real-time channel interfacing to the real-time subsystem **230** has a unique identity, and is treated by the system as if it were physically terminated at the real-time subsystem **230**. In practice, however, the number of physical connections between the telephone room subsystem **206** and the real-time subsystem **230** is minimized by multiplexing groups of 30 channels onto full-duplex, high-speed synchronous links **232–235**.

TDM is used on each of the two links connected to a real-time processor **410, 412**. Each link provides 8000 frames per second with each frame 125 microseconds in duration. The 125 microsecond frame is further subdivided into 32 time slots with each slot able to sink an 8-bit data sample. At the incoming switching point of the system **100**, each digitized voice channel deposits a voice data sample of eight bits into a specific, assigned time slot in a frame which remains its own unique slot in each frame until voice sampling ends. Thirty time slots in each frame carry voice data while the remaining two time slots in each frame are used for control and synchronization data. The control slot allows control data to pass up and down the link between the real-time subsystem **230** and the outboard multiplexing equipment, which interfaces the physical channels onto the high-speed link. The control slot carries down the link the individual control message generated as a result of supervisory commands given by an application process. Tne reverse direction carries status and result data back to the system **100**. The synchronization channel is used to keep the multiplexors and interface boards in step with the switching hardware of the real-time subsystem **230**

The time allocation on the high-speed synchronous link conforms to CCITT recommencations for 2.048 Mbps PCM links carrying 64 Kbps encoded voice. However, in the real-time subsystem-telephone room subsystem connection, a unique control protocol exists that is specific to the various line interface boards and line group controllers of the telephone room subsystem **206**, and does not conform to that of the CCITT recommendations since this also encompasses the commands that are sent over the control slot. The differences between the two protocols are such as to be easily handled by logic on an interface board were it ever desired to directly connect a CCITT 2.048 Mbps link to a real-time system. Of the 32 channels in a link, channel **1** carries synchronization data, channel **17** carries control data and the rest carry voice data.

The telephone room subsystem **206** (See FIG. **3**) is outboard of the real-time subsystem **230** attached via the high speed serial links **232, 233** and provides a direct interface to individual analog telephone lines, trunks, and other voice-grade circuits. The telephone room subsystem **206**, as it relates to the real-time subsystem **230**, consists of the following:

(1) A variety of line interface boards **310–318**, each interfacing two physical voice-grade circuits of the same type. The line interface boards provide analog-to-digital and digital-to-analog conversion (via mu- or A-law codes), together with generation and detection of supervisory signals such as dial-tones, dial-pulses, on-hook off-hook condition detect, etc.

(2) Two line group controllers **302, 306** are associated with each physical card cage capable of housing 15 line interface boards **302–318**. The 15 line interface boards

provide 30 individual, full-duplex real-time channels, multiplexed by one of the line group controllers **302, 306** onto a single, 2.048 Mbps serial link connected fo the real-time subsystem **230**. Each line group controller **302, 306** is microprocessor-controlled (Intel 8085) and receives control and supervisory data over the control channel slot from the real-time subsystem **230**. Each line group controller **302, 306** also collects status information from the line interface board **310–318** and sends it to the real-time subsystem **230** via the control slot on the up-link. Selection of one of the two line group controllers **302, 306** as controller is under software control via the real-time subsystem **230**

Each of the different types of line interface boards is tailored to the operational characteristics of a particular class of telephone line, trunk, or voice-grade circuit. All, however, have a basic set of features:

(1) Termination of two voice-grade circuits of a type unique to a line interface board.

(2) PCM encoding and decoding for each line.

(3) Dual port access to two different line group controllers.

Tne PCM encoding/decoding function is performed by Codecs (Coders/Decoders) on the line interface board. A Codec performs an eight-bit, non-linear analog-to-digital and digital-to-analog conversion. This non-linear (quai-logarithmic) conversion is used since it is more sensitve to the lower end of the voice intensity spectrum where the major part of speech information is found. The difference between the transfer characteristics of mu-law used in America and the A-law used in Europe is slight and easily converted from one to the other.

The quasi-logarithmic Codec output of one 8-bit floating point number for each voice sample occurs each 125 microseconds as a direct consequence of the sampling rate, which is 8000 samples per second per voice channel. This floating point format of each PCM sample permits a much greater sound intensity range to be encompassed than would be possible with a strictly linear (for example, integer) representation.

Four classes of line interface board are necessary for most telephone applications. An operator line interface board **310** provides two simple four-wire links to operator station headsets with no DTMF, supervisory or control signaling. A concentrator line interface board **312** terminates two "dry" (no line voltage) two-wire links from remote line concentrators with no supervisory or control signaling other than DTMF. A direct inward dialing line interface board **314** terminates two 2-wire DII trunks, handles all DTMF signals and detects supervisory and control signals and incoming dialed digits. A loop-start/ground-start line interface board **316** terminates two pay station telephone number conventional lines, can accept incoming calls, can seize a line to dial outgoing calls, and can handle all DTMF, supervisory and control signals.

The dual-ported line interface boards **310–318** connect to the pair of eight-bit data highways **304, 308**, each attached to a separate line group controller **302, 306**. Either line group controller of a pair **302, 306** can multiplex up to four operator line interface boards **310**, and up to 11 line interface boards **312–318** of any other type onto a high-speed full-duplex synchronous link interconnecting the line group controller **302, 306** to the real-time subsystem **230**. The line group controllers **302, 306** sequentially poll each line interface board **310–318** for channel data and status conditions and pass to each line interface board PCM data for output, and any commands to set line conditions or requests for status. If a line interface board fails the two telephone lines attached lose services. Line group controllers and line interface boards are redundantly connected so that any voice channel can be routed into the real-time subsystem **230** via two independent paths. The executive control processes in the real-time subsystem **230** affect a switch to the other line group controller/synchronous link of a pair if the active link or line group controller is detected as failed. In brief, of the two dual-ported line group controllers **302, 306** per 15 line interface boards **310–318**, one is the active controller and the other is a hot stand-by.

A telephone room subsystem **206** consists of from one to seven racks, each containing from one to six card cages and redundant power supplies. Each card cage holds two line group controllers, up to four operator line interface boards, and up to 11 other line interface boards. Altogether, a telephone room subsystem **206** can accommodate up to 1260 voice-grade circuit terminations. The system **100** can support four separate real-time subsystems, each with its own telephone room subsystem permitting a total of 5,040 voice-grade circuit terminations.

E. Hardware Architecture

FIG. **4** is a representation of the real-time subsystem **230** architecture, whose main components consist of the following:

(1) Two high-speed TDM real-time buses **414, 416**, each of which is a 16-bit parallel synchronous link used for the interchange of PCM or other data between ports on the link. The two buses are asynchronous relative to eacn other.

(2) Two real-time executives **406, 408** with each in control of the switching and routing hardware of one of the two TDM buses **414, 416**. The executives **406, 408** host system software processes which provide a simple, logical interface to the rest of the system **100**, including application processes.

(3) External transfer switches **402, 404**, interface serial PCM or other data links to the TDM bus pair. The external transfer switches **402, 404** and TDM buses **414, 6** are used to exchange data between real-time data channels, which in voice applications are the telephone room subsystem channels and the real-time processors **410, 412** in the system **100**.

(4) Up to 22 real-time processors **410, 412** interface with both TDM buses **414, 416** and serve as hosts to dynamically allocated software transient processes, thus providing a capability for a variety of data signal processing and, in voice applications, voice data compression and decompression.

The real-time subsystem **230** is fault tolerant and can survive single failures of either its executive processors buses, or any switch on any real-time processor **410, 412**. In the event of failure, channels are rerouted via alternative resources. The real-time subsystem **230** uses load-shared redundant capacity.

Internal to each real-time subsystem **230** is an independent bus-pair **414, 416** denoted S and T. Each bus is functionally identical and backs up the other so that if one fails. the real-time subsystem **230** can still operate normally using the survivor. The function of the buses is to exchange real-time data among external transfer switches **402, 404**, real-time processors **410, 412**, and real-time executives **406, 408**.

Each TDM bus **414, 416** is 16 bits wide and of the 512 time slots per 125 microsecond frame available, four slots (**0, 1, 2,** and **3**) are preassigned by the system to exchange control information between the various devices that are attached to the bus. This leaves 508 usable slots for real-time data channels. Each time slot provides a 0.244 usec period during which PCM or other data samples can be put on a bus from one source and read from the bus by any number of destinations. Each time slot occurs once per frame, and frames recur at 125 microsecond intervals. Thus each slot permits 8000 PCM samples (or other data) to be transferred per second.

For voice applications, an 8-bit PCM speech sample occupies one time slot and one slot in the frame is assigned for each speaker when a pair of channels are interconnected. This means a real-time subsystem **230** could support as many as 500 simultaneous oonversations, if no slots were required for DTMF or other signals, or for supervisory control.

Associated with the 16 physical data lines of each TDM bus **414, 416** is a parity line, a number of control lines which supply the basic frame and slot reference signals to the devices using the bus, and an ACK/NAK signal line used in the exchange of non-voice data to signal the detection of parity errors on the bus. If the sender receives a NAK indicating detection of a parity error, that data is sent again in the same slot of the next frame. If a number of retries occur and the parity error continues, the bus is deemed to have failed and all traffic is rerouted onto the surviving bus. Parity is not tested on PCM transfers.

Or the two real-time executives **406, 408** in the real-time subsystem **230**, one is designated as primary by virtue of its being the first real-time executive to become operational at system start-up. This prime real-time executixe originates the frame and slot clocks for both the S and T buses. Each bus operates synchronously but relative to each other the S and T buses operate asynchronously. The second real-time executive regenerates the timing clocks synchronously from the primary's clocks. The primary's clocks can be strapped to originate either from a self-contained crystal or from an external "master clock".

Attached to the S and T bus-pair **414, 416** are up to 42 external transfer switches **402, 404** which perform two major functions. (1) They interface and control the duplex 2.048 Mbps serial links connecting the real-time subsystem **210** to the data channels and in voice applications, to tne telephone room subsystem **206**. (2) They control the data transmission into and out of the 512 time slots on each TDM **414, 416** bus (S and T buses) from the data channels connected to the input of each external transfer switch **402, 404**. In voice applications, there are 64 channels attached to each external transfer switch since each external transfer switch has two serial links, each having a capacity for 30 multiplexed PCM channels and two control and supervisory channels. Data interchange is defined by the contents of the S and T port memories in each external transfer switch **402, 404**, with each memory containing 512 port command words. These words control data transfers from data channels into the external transfer switch **402, 404** and in voice applications, from any of the **64** multiplexed PCM channels onto specified time slots of the available 512 on either the S or T bus.

The external transfer switch **402, 404**, together with the real-time buses **414, 416**, form a totally autonomous switching system. Once the controlling real-time executives **406, 408** have loaded the external transfer switch command memories to reflect a desired channel interconnect, data is transferred through the external transfer switches **402, 404** and along the bus, driven only by the synchronizing bus clock. No intervention or processing by the real-time executives **406, 408** is required to maintain a given interconnect. Any channel interfaced to an external transfer switch via a serial link can be connected to any other connected channel by assigning appropriate time slots, and properly loading corresponding commands to the respective external transfer switch port command memories.

Each external transfer switch **402, 404** is physically identified by a seven-bit S and T bus identifier (STBID), associated with the physical connector into which the external transfer switch **402, 404** is inserted. The STBID is reset by the controlling real-time executive **406, 408** to designate which external transfer switch is being addressed when control data is transferred from a real-time executive to an external transfer switch (during the four-slot bus control period).

Each real-time subsystem **230** contains two real-time executives **406, 408**. Each real-time executive **406, 408** interfaces to both the S and T bus **414, 416** and is responsible for managing one of the two. Thus, they are referred to as real-time executive-S **406** and real-time executive-F **408**, respectively. Real-time executive S **406** is the processor that activates the various control signals associated with the S-bus **414**, and can load the command memories associated with the S-port of each external transfer switch **402, 404**. The one exception to the control of the T-bus **416** by real-time executive-T **408** and the S-bus **414** by real-time executive S-**406** is in the frame and slot clock signals for both the S and T buses **414, 416**, which are under the control of the prime real-time executive (defined as the first real-time executive to become operational at system wake-up). The other rea**1**-time executive regenerates the bus clock signals in synchronization with the prime's clock. Similarly, real-time executive-T **408** controls the T-bus **416** and its associate external transfer switch bus ports.

As shown in FIG. **6**, a real-time executive **406, 408** includes:

(1) A standard processor module **500**, with resident executive and x and y inter-processor executive services buses **912, 914**.

(2) A real-time processor extension **604** providing an additional 24K words of program memory **614**, 64K words of data memory **616**, and a microprogrammed DMA controller **618**.

(3) An internal transfer switch **606**, similar to an external transfer switch **402** except that instead of interfacing serial lines to the S,T bus, it interfaces the real-time executive's data memory to the S,T bus via **17** logical DMA channels **620**.

(4) A jumper clip manually placed on the S or T bus backplane next to the connector into which the internal transfer switch **606** is inserted. This clip distinguishes the real-time executive-S **406** and real-time executive-T **408** from a real-time processor **410, 412**.

The real-time processor extension **604** provides 24 words of added program memory **614**, 64K words of data memory **616**, the DMA channels **620**, and also extends the normal instruction repetoire of the standard processor module **500**.

The real-time processor extension **604** can generate interrupts to its associated standard processor module

602 when its data memory buffers either fill or empty after data transfer to, or from the real-time slots on the S and T bus 414, 416. This is the general method of communication from the DMA controller 618 in the real-time processor extension 604 to processes in the standard processor module 602.

It is possible to set an interrupt mask register by one of the real-time processor extension special instructions to enable or disable specific interrupts as well as switch on or off certain real-time processor extension facilities. The conditional test instructions, provided in the real-time processor extension set, can be used to decode the specific cause of an interrupt. A First-In, First-Out (FIFO) buffer between the real-time processor extension 604 and the standard processor module 602 allows multiple interrupts to be posted to the standard processor module 602 processor by the real-time processor extension 604.

A real-time processor 410, 412 is physically identical to a real-time executive 406, 408. It provides essentially the same functions, the only differences consisting of the following:

(1) The 17th DMA channel, used in a real-time executive 406, 408 to send control information onto the bus during the first four slot times, is disabled in a real-time processor 410, 412.

(2) The software processes, hosted by the real-time processors 410, 412, are not involved in the management of the real-time subsystem 230. Processes in a real-time processor 410, 412 use the services and facilities of the real-time subsystem 230 in ways comparable to the use of processor facilities by processes in other system 100 processors. The important difference being that the real-time processor 410, 412 can directly and rapidly acquire and generate real-time data by virtue of its physical connection to the real-time bus.

A real-time processor 410, 412 can host predefined transient or user-defined system processes programmed in standard processor module assembler language. A real-time processor 410, 412 process node could be subordinate to a controlling primary job process resident in a general purpose processor 942, 944, or could itself be a job's primary node invoked in a real-time processor 410, 412 for time-critical applications.

F. Software Architecture

The real-time subsystem software architecture illustrated in FIG. 11 shows the major software components resident in both primary and secondary real-time executive and their relation to other processes. Within a real-time executive 406, 408, several major functional areas exist.

A prime supervisor process 1102 performs a variety of executive and utility functions including interfacing requests for assignment from the system device manager (SYSDEV) 1104 in response to acquire requests from user application process 1106. When the real-time subsystem 230 powers up, the first real-time executive 406, 408 processor (either S or T) to report itself ready is assigned as primary real-time executive. Assign requests from SYSDEV 1104 are always passed to the prime supervisor first. The prime supervisor then decides which of the two real-time executives 406, 408 should manage this particular assignment and all subsequent requests relating to it (as identified in future requests by the owner's job number which is passed with all requests). This decision is made with an aim to load-share requests between the two real-time executives 406, 408. Having decided which real-time executive is responsible for this job's channels, the information in the assign request is recorded by prime and passed to the secondary real-time executive supervisor. The secondary supervisor also records the data and acknowledges the request back to SYSDEV 1104. Subsequent requests to the other processes in the real-time subsystem 230 are directed to the primary real-time executive first, and then redirected to whichever real-time executive is responsible to the requesting job for its channels.

The user does not communicate directly with the supervisor 1102. This link is managed via SYSDEV 1104 which also sends free requests to the supervisor 1102 when a user wishes to release a channel after it is no longer required.

Switch circuits (circuits) 1108 respond to a user request to create and destroy interconnections among channels owned by the job. Packets, sent to circuits 1108 as a result of switching requests from the channel owner, contain simple interconnect maps describing which data source channels should be connected to selected destination channels. A particular subset of interconnections can be uniquely identified by reference to a circuit network ID, a number supplied by the user 1106 when setting up an interconnection. An interconnection can then be destroyed with a single request referencing that same circuit network ID. A user can specify any number of circuit networks (up to 255) using nonoverlapping channel sets, provid-d that all of the channels are "owned" by the job.

Data switch and event notification (DSEN) 1110 manages information passed along the control channels of the serial links connecting the external transfer switches 402, 404 to the external subsystem, which multiplexes and interfaces individual physical channels to the system 100. Data switch and event notification 1110 provides a simple interface to these control channels. A user wishing to forward control data to an owned channel sends a request to data switch and event notification 1110 identifying the channel, together with the control command code and an associate argument. Data switch and event notification 1110 routes this message along the appropriate serial link control channel to the line group controller 232, 233 which controls the designated channels' interface board.

Unsolicited status messages originating from a channel are collected by data switch and event notification 1110. If the channel is owned by an active process, data switch and event notification 1110 passes the control message to the owner. If the channel is not owned, the message is passed onto the system device log-on process (SYSDLO). SYSDLO then decides if some action should be taken, possibly initiating a job to respond to the message. On a system wide basis, an unsolicited stimulus message from data switch and event notification 1110 is an external event with SYSDLO acting as the unsolicited event halder. The following describes an example of data switch and event notification 1110 operation for a voice message call sequence.

Event 1—A ring is detected. A line group controller 232, 233 detects the line state change via a line interface board signal, and sends a message to data switch and event notification 1110 via Channel 17, the S,T bus, and a DMA data channel into the real-time executive memory.

Event 2—Data switch and event notification 1110 consults the channel assignment tables and determines that this channel is owned by SYSDLO (that is, it is not assigned to an application job). Data switch and event

notification **1110** passes a message containing the line ID to SYSDLO.

Event 3—SYSDLO's tables describe what action to initiate if this event, or sequence of events occur on this channel. In this case, it requests JSAM to initiate a voice messaging process to handle the call.

Event 4—JSAM starts a voice messaging process in an ideal machine, passing the ID of the channel as a start-up parameter.

Event 5—Voice messaging "acquires" the channel with a request to SYSDEV **1104**.

Event 6—SYSDEV **1104** "assigns" the channel in response to the "acquire".

Event 7—Voice messaging sends a control packet to data switch and event notification **1110** to set ring back on the line.

Event 8—Data switch and event notification **1110** passes the message to the line group controller **232, 233** controlling the line interface board **314**, which turns on the ring back generator in the line interface board **314**.

Event 9—The phone is answered by the system **100**.

A signal generator (SIGGEN) **1114** is a utility function that permits the real-time subsystem **230** to generate a tone, or set of tones and attach the tone to a specific circuit network. A tone may be in the range of 1 to 3200 Hz at integral frequencies, subject to quantization approximations A single request to SIGGEN **1114** can specify the channels to receive the tone frequency, and relative amplitude of the tone together with the tones' duration.

Dialer **1116** is a utility function used with telephone applications. A process that has acquired a channel can request that either DTMF or dial-pulses be generated on that channel provided the channel has the correct type of line interface board to activate the request.

With DTMF dialing, DIALER **1116** uses SIGGEN **1114** to send DTMF tones to the channel being dialed in accordance with CCITT recommencation Q.23 (each digit tone is transmitted for approximately 50 ms, with an inter-digit silence of approximately 50 ms).

To generate dial-pulses, DIALER **1116** uses data switch and event notification **1110** to send dial-pulse control requests along the normal supervisory control channel to the line interface board **314** to which the channel is attached. The line interface board **314** then switches the line condition to implement each pulse.

The record and playback functions in voice applications can send data samples to a disk dataset with delta data compression. This initial data compression can be further enhanced by replacing strings of identical samples (representing silence in voice data) with a run count. Using both compression and run-length encoding, overall data reduction of more than two to one can be achieved for normal speech The compression algorithm uses an error-correcting feedback loop, with hardware look-up tables to speed the loop delay It operates as follows:

(1) The compression technique uses a feedback algorithm to reduce the error between input and the regenerated samples An incoming 8-bit PCM sample has a feedback 8-bit sample subtracted from it to yield an 8-bit result. The feedback sample represents the reconstructed previous linear PCM sample derived from the output of the compression algorithm.

(2) The 8-bit difference sample is used to address a 256 word five-bit look-up table using the real-time processor extension special COMPRESS instruction This look-up returns a five-bit compressed difference signal.

(3) Three successive compressed difference samples are packed into one word and bit zero is flagged to indicate the format of the word. Such words are blocked and then sent to the dataset.

(4) Each difference sample is expanded back to an 8-bit form by the EXPAND instruction of the real-time processor extension, using another hardware look-up table It is then accumulated with previous reconverted samples to form the normalized feedback sample for the next differencing operation.

Because the feedback algorithm prevents accumulative errors, the compressed PCM output is far more accurate than a simple five-bit sample would suggest.

To reconstruct the original signal, the five-bit samples are expanded and integrated since drift in the difference signal has been eliminated during record. For further reference to this modulation technique, refer to "Delta Modulation Systems", R. Steele, Wiley & Sons, 1975.

Finally, certain markers are placed in the data stream. A data stream marker is a three-word sequence, with a control word indicating a data marker followed by a unique mark ID. This ID can be used by processing routines to perform data processing functions on the dataset. During record, a data stream mark is inserted into the recorded data at one second intervals followed by an End-Of-Message (EOM) marker at the end of the data stream.

KERNEL SYSTEM

The Kernel of the system **100** includes the set of 8 to 32 processors, organized into subsystems of from four to six types, plus the system-level software and data base necessary to integrate the subsystems and provide the user or application programmer with an apparently monolithic, powerful and easy-to-use computer.

A feature of the system **100** is the automatic initial program load (IPL) when a system, or any processor, is started or restarted; another is the ability to dynamically maintain, on-line, the system files required for "SYSGEN". A third, and even more important, feature is the distribution of system functions among the processors to achieve true parallel processing.

A. System Resource and Job Management

Job scheduling, allocation, and management resides in primary form in one executive services processor **916** or **918**, and in secondary form in the other. JSAM both actively and passively maintains the status of all processors in the system, the performance of each, and the resources of each currently unassigned and available. If a new processor is reported by interprocessor communications, JSAM uses the system configuration file to sequence the integration of the processor into the system. Upon being integrated, the new processor becomes a set of resources and functional capabilities available for use.

If a processor fails ortraps, JSAM causes the processor to be logically deleted from the system and reconnected under a new identity. JSAM also effects the relocation or recovery of those processes which had been initiated or allocated by JSAM. JSAM receives job request packets from any qualified originator, assigns job numbers, allocates any necessary resources, and either initiates a new process or hands the job request packet over to an appropriate virtual machine operating system.

JSAM responds to requests from any process desiring to set up or subsequently tear down a process-net: a set

of one or more processes, initialized to establish any necessary intra-net communication, allocated to the requesting process and controlled much as if the net were a pseudo-device. In addition, JSAM maintains a TICKLER file, used to schedule periodic and deferred job requests on intervals of six minutes to one year, with a resolution of six minutes.

The system **100** is designed to respond to outside stimuli. Stimuli from devices not allocated to some active process (such as the ringing of a previously quiescent telephone line or powerup of an operator station) are sent to SYSDLO for analysis and resolution. SYSDLO uses data from the system device file to determine the action to be taken at this time of day and day of the week, for the specific device. Actions to be taken can include:

(1) Wait for additional stimuli (e.g., rings) before responding.

(2) Log an illogical stimulus and optionally cause the device to be logically turned off.

(3) Actively interact with the device (e.g., initial log on of an operator station) to define the specific nature of the required response.

(4) Determine that no action is required.

(5) Initiate a job request packet to JSAM, with parameters obtained from the SDF record of the device.

SYSDLO is a singular process which may reside anywhere in the system where adequate data memory exists for the required on-line data base.

B. System Files and Device Management

Each system **100** is required to have two disk packs, pre-initialized as system disk packs **930, 932** (SYSDISKs) and mounted on drives attached to two different disk data processors **934, 936**. Each SYSDISK includes special transient routines to permit autonomous wakeup of the disk data processor **934, 936** under certain circumstances. In addition, each SYSDISK **930, 932** contains a logical duplicate of the following files:

(1) SYSCAT—The system catalog is maintained by SYSDIR as data sets are created and deleted. SYSCAT includes all disk data sets, the data sets mounted on each magnetic tape unit, and may include unmounted magnetic tape data sets. Attributes of each cataloged data set include: (a) Creation date and creator identity, (b) Access security specification, and (c) Redundancy type and location. The following redundancy type and location information is provided: (a) singular—volume ID (or IDs if more than one volume), (b) duplex—volume IDs, copies one and two (recorded in parallel but not automatically re-duplexed if one volume fails), and (c) duplicate—volume IDs, copies one and two (automatic recreation of second copy if one copy fails and is not recoverable).

File organization options include: (a) record length (maximum if variable), which can be fixed or variable, (b) direct or indexed (number of directories), (c) offset and length, index one, each record, and (d) offset and length, index two, each record.

A name is also associated as an attribute of each dataset.

(2) SCF—The system configuration file is maintained interactively via SYSMON. This file lists the minimal set of hardware and software required for processor, subsystem and system operation. It is used by JSAM to effect the wakeup of processors, subsystems, and the entire system, and to facilitate recovery in the event of a failure. This file initializes SYSMON for the log data to be captured and summarized on-line.

(3) SDF—The system device file is maintained interactively via SYSMON and indirectly by service processes in virtual machine operating systems that permanently "own" devices (such as the telephone answering support system, which permanently "owns" most of the attached telephone lines). For each device, this file includes: device type and subtype or subtype range, virtual machine operating system ownerhip, if any; account ownership, if any; a list of system termination points, by channel type and physical address, in/-out of service, data and time, physical location, extra system identifiers (telephone number, unit serial number, etc.), periodic diagnostics (e.g., line check) parameters, service schedule by time-of-day, day-of-week, and type of stimulus versus type of response.

(4) The kernel program library is maintained by the kernel program librarian, through SYSMON. The library of all programs directly executable from bipolar program memory/writeable control store by any processor in the system. Included for each program is the set of resource and precedent programs necessary for execution. Each program is indexed by name, version and change level.

(5) The TICKLER file is maintained by JSAM. It contains job reqjests scheduled for activation from six minutes to two years in the future. Each request can include parameters to permit automatic rescheduling at intervals of minutes, hours, days, weeks, or months.

SYSDIR exists in a primary and secondary form, one to each disk data processor **934, 936** which hosts a SYSDISK **930, 932**. SYSDIR mechanizes the following functions:

(1) CREATE, which allocates space on a requestor-selected or SYSDIR-selected volume, and catalogs the data set.

(2) DELETE, which decatalogs a dataset and restores to available space all space allocated to the dataset.

(3) RECATALOG, which effects a change in the name or other attributes of a dataset catalog descriptor.

(4) OPEN, which establishes the linkage (headers) for communication between the requestor and the disk data processors hosting the dataset.

(5) CLOSE, which terminates the linkage established by the OPEN.

SYSDEV manages the inventory of external devices attached to the system **100** including telephone lines, operator stations, pointers, array processing unit channels, data links, concentrators, and any others. SYSDEV, in conjunction with SYSMON, maintains the system device file. Any synchronous bus processor or interactive services bus processor, upon wakeup, communicates with SYSDEV to obtain the type and identity of all attached devices. In addition, SYSDEV mechanizes the following functions:

(1) CREATE, which effects the posting of an additional device.

(2) DELETE, which effects the posting of the removal of a device.

(3) RECLASSIFY, which changes designated parts of the record of an existing device.

(4) GETDEVICE, which allocates a device by identity or by type and list of subtypes.

(5) RELEASE, which returns to inventory a specifid device, all devices owned by a process, or all devices owned by a job.

C. Utility Software

System Logger (SYSLOG) is initiated and allocated by JSAM to reside in a disk data processor **934, 936**. It receives all log packets and validates them for format and generation rate. It posts each log packet to an appropriate FIFO file on disk for subsequent processing. If desired, the disk files may be moved to tape.

Kernel Program Librarian is initiated on demand via a SYSMON file update console. Used to post new, or update old, programs (from levels **0** through **4**) in the kernel program library. Sources can be any of various I/O devices or files created by the system assembler running under the program development system.

Disk Utilities are accessible via SYSMON and respond to the commands DUMP, COPY, PACKINIT, and VALIDATE.

Voice Compression and Record is a pseudo-device invoked and acquired by a process (job) desiring to acquire and store a voice message. The program resides in a data processor on the synchronous bus **912, 914**. During initiation it sets up to record the message through one or two (selectable) disk data processors **934, 936**. The message index and any segmentation marks are supplied by the requestor.

Voice Message Retrieval is a pseudo-device invoked and acquired by a process (job) desiring to retrieve and playback a recorded voice message. During initiation, the requestor supplies the message ID or IDs (if two exist). Datasets are opened as necessary and playback is effected under supervisory control of the requestor.

Voice Response Unit playback is a psuedo-device similar to voice message retrieval, except that the program is given the voice response unit file ID and a string of voice response unit segment IDs which it uses to construct the voice response unit sequence and autonomously terminate.

System Program Loader (SYSLOADER) is invoked only by REX in the target processor, as the result of a request packet. It provides read-only access to the kernel program library, and effects program relocation where required.

D. Virtual Machine

A virtual machine may be created to host the processes of a new job or to provide parallel processes for an existing job. At the time of creation, a nucleus of real memory is allocated. In the virtual machine engines, the following rules are applied:

(1) Instruction memory is partitioned from data and stack memory as a separate space.

(2) Instruction memory is read only after the program has been loaded.

(3) Memory is not swapped or moved to bulk storage except as explicitly requested by the program executing in the virtual machine.

(4) Memory used for stack or data space may be dynamically managed or completely allocated during virtual machine initiation.

(5) Interactive jobs initiated from a terminal (or from an operator station for so long as no telephone line is in use) are always assigned a lower priority than jobs involving a telephonic line.

(6) System-initiated, periodically scheduled jobs not involving a telephone line, and not operating interactively, are assigned an even lower priority.

The resources acquirable by a single virtual machine are subject to the following limitation:

(1) Memory—Each partition (instruction, data, stack) has a 24-bit byte addressing capability, mechanized as an 8-bit segment register (most significant) and a 16-bit address register (least significant). The most significant 12 bits address a page table to access the most significant 11 bits of the sought real memory address. The total memory used by one virtual machine can extend to 8 M bytes; allocation is dynamic.

(2) Devices—Virtual devices are acquired by an application program from virtual machine monitors as required. A virtual machine monitor acquires corresponding real devices from the system as necessary. The maximum number of virtual devices acquired by a virtual machine is determined at the time the program running in the virtual machine is loaded.

(3) Files—Unless special steps are taken upon creation, files are accessible only through the virtual machine operating system under which the files were originally created. Furthermore, the virtual machine operating system can categorically limit, through virtual machine monitor parameters, the space occupied and the number and types of files to be created by a virtual machine.

VIRTUAL SYSTEMS

At the user level, a DELTA system consists of an arbitrary number of virtual machines. Each virtual machine uses a virtual machine operating system to fulfill the operational purpose for which it is created, such as on-line interactive operations, application program execution, etc. A virtual system consists of:

(1) A virtual machine composed of a virtual machine interpreter and a virtual machine monitor.

(2) A virtual machine operating system.

(3) A set of real devices allocated upon request of the virtual machine operating system or by the application program through the virtual machine operating system.

(4) A set of files accessible (in a security sense) to the virtual machine operating system, opened either by virtual machine operating system or by an application program through virtual machine operating system.

Three generic types of virtual system are presently defined, each with its own virtual machine operating system.

System Monitor (SYSMON) is an interactive system via which a variety of specialized monitors can be invoked. The specialized monitors include (1) a maintenance monitor, (2) a system programmer monitor, and (3) an operations monitors, one for each unique VMOS. Within each specialized monitor, SYSMON provides the ability to identify a set of logger events, plus a set of corresponding functions to permit on-line access to system and application performance and status data. Examples might include:

(1) The number of operator stations of certain subtypes presently logged on, and the identities of the operators.

(2) A running summary of reported error conditions.

(3) A printed log of alarm conditions.

The psuedo-machine operations accessible to the SYSMON virtual machine interpreter include privileged instructions not accessible to other virtual machine interpreters. Thus, SYSMON is the only virtual machine operating system able to interact at the level of the writeable control store within the system **100**. To preserve access security to the maximum possible extent, activation of a SYSMON virtual system is restricted to a set of physical operator stations so desig-

nated in the system device file. In addition a user ID and password validation is required. In addition to the monitor identified above, SYSMON also provides, via a file update mode, the only means for updating the kernel system files.

The Program Development System (PDS) is a software system derived from the Pascal system of the University of California at San Diego (UCSD). Each virtual machine created under the program development system appears to be a single-user data processor with a. terminal, a printer, and a set of files accessible to the user. The user may access any of the following:

(1) A File Handler.

(2) Two Editors; one screenoriented and one line-oriented.

(3) Pascal Compiler.

(4) BASIC Compiler.

(5) Linker for the Pascal and BASIC compilers.

(6) Processor assembler and linker.

(7) Debug utilities.

(8) Computational and service utilities.

The user may also invoke any program created under PDS which executes on the PDS virtual machine engine.

Telephone answering support system (TASS) provides the functional framework for the telephone service system communications services centers. Telephone answering support system is a stimulus-response system incorporating a highly-structured, very versatile telephone receptionist interface. Telephone answering support system integrates the circuit switching and signal (i.e., voice) processing to provide for the acquisition and retrieval of voice messages, using digital disks as the storage medium. Telephone answering support system coordinates the assignment of operators to calls, control of the telephone network, maintains the TASS data base, and the acquisition of data for SYSMON supervisory use.

The telephone answering support system data base consists of:

(1) A set of account-ordered files, including formatted display data, accounting records, and messages and bulletins.

(2) An operator records file.

(3) The service schedule and other portion of the records for those devices (lines, trunks, etc.) flagged in the system device file as being updatable via the telephone answering support system.

(4) The telephone answering support system event log SPOOL.

Services provided under telephone answering support system include:

(1) Call intercept (Incall) handling, including message acquisition, voice, or typed.

(2) Retrieval call (Recall) handling, including MFT and voice command control of playback.

(3) Message dispatch (Outcall), the scheduled or unscheduled (demand) active delivery of a message.

(4) Field service dispatch.

(5) Order-taking.

(6) Classified ad voice response.

INTERACTIVE SERVICES SUBSYSTEM

A. Overview

The interactive services subsystem **252** (FIGS. **2**, **7**) within the system **100** provides executive services and management of data transfers betwen processes and various attached peripheral devices. The system **100** may contain from one to four interactive services subsystems **252**, each processing two interactive buses **704**, **708** and two dedicated interactive services executive processors, **702**, **706** individually managing a separate bus. Each interactive services subsystem **252** parovides up to 64 device controllers through which a variety of peripheral devices such as terminals, magnetic tapes, pointers, etc. may be attached. (Disks are controlled and managed by the file services subsystem **908**). Thus up to 1024 asynchronous full-duplex serial lines can be attached to a single interactive services subsystem **252** and for each a total of 1984 data buffers may be defined. Since each interactive services subsystem **252** has its own complement of processing resources, data buffers, buses, controllers, and handlers, a smooth growth path without bottlenecks is assured with the capacity to support ultra-large configurations.

Each interactive services subsystem **252** controls a separate bus pair to which is attached dual-ported, fully-buffered device controllers. The interactive services executives **702**, **706** use an adaptive polling technique and a multiplexed DMA channel controlled by a separate, independent microprocessor. The microprocessor transfers data between memory buffers and devices at a rate determined by the demands of the various devices.

The interactive services executive processors **702**, **706** used are members of the standard processor familty, which consists of a 16-bit standard processor module **500** (**724**) and an interactive processor extension unit **726** that provides data buffering and bus control functions. Each interactive services executive **702**, **706** hosts multiple concurrent processes, (each process generally a Level **1** element of the kernel system), provices a device and channel handler functions, and are part of the system software. An interactive services processor **702**, **706** can be user-programmed to augment the kernel system when a device, unique to an application, is attached to the interactive services subsystem **252**.

However, since interactive services processors **702**, **704** are programmed in the low-level SPM assembler, the resulting process coexists with other processes in an unprotected environment and programming the interactive services subsystem **252** becomes a privileged task, requiring an understanding of the standard processor module **500** hardware, resident executive, and all other kernel system protocols.

A Level **3** Pascal-programmed application task uses the services provided by an interactive services subsystem **252** through procedure calls. These procedures are a subset of the virtual machine monitor's intrinsic support functions which handle general input/output to devices and datasets. To an application, a device attached to the interactive services subsystem **252** appears as a channel along which data and control information can be passed. A well-defined protocol dictates the sequence of interactions that can take place between an application and a device and is identical to the protocol used in handling data channels from datasets residing on disk or magnetic tape. Many of the procedures used for data transfer are the same for both device-related and dataset transfers. Magnetic tapes are attached to the interactive services subsystem **252** as peripheral devices, but logically support datasets. Therefore, dataset-related calls from an application, program can be directed to either the file services subsystem **908** or the interactive services subsystem **252** depending on whether the dataset has been opened on disk or magnetic tape.

Access from an application residing in the information processing subsystem **906** to the interactive services subsystem **252** is via Level **2** manager processes which manage (on a system-wide basis) device allocation and usage for single or multiple interactive services subsystems **252**. This management function is essential in multi-task environments, where many concurrent and simultaneous applications need to acquire and use a fixed number of peripherals. To acquire a terminal, printer, or line, a request to the system device manager (SYSDEV) must first be made. SYSDEV allocates the device; builds a communications path between the application and the device's handler, and records the details in its internal allocations tables. Similarly, a magnetic tape dataset is opened through the system directory manager (SYSDIR) which acquires a magnetic tape drive, ensures that the correct tape is mounted, and builts a communications path from the application to the magnetic tape handler in the interactive services subsystem **252** or subsequent data transfer operations.

The interactive services subsystem **252** does not generally host any kernel system Level **2** processes. The processing capacity is reserved for the various device-related processes that need the specific environment provided in the interactive services processors **702, 706,** namely the unique access to the data buffers and control registers of the physical channel controllers attached to the interactive bus **704, 708.**

B. I/O Protocol

Irrespective of the nature of a device, a general communications protocol exists between a using process and a device. This protocol is consistent not only for applications programmed in Pascal and running in the virtual machine environment in the information processing system **906** but also for low-level standard processor module assembler processes using basic resident executive services. The protocol is composed of the following distinct steps that lead up to, implement, and then terminate a data transfer:

(1) Acquire the device for use.

(2) Open a communications path to the handler.

(3) Access the device for read or write.

(4) Transfer data between the user and the device.

(5) Terminate a data transfer operation.

(6) Close the communications path to the device handler.

(7) Release the device back to the system.

(8) Between the acquire and release stages, send and receive control information to the device as appropriate.

These steps do not have to be sequentially performed for each data transfer. Often a device is acquired and opened for the duration of a job's existence and in other instances, only control functions have any logical meaning when related to a particular device. However, all of these cases are contained within the general protocol.

Before being able to talk to a device, the user-process must first own it. An acquire request is routed to SYSDEV either requesting a specific device (identified by its internal system ID), or a type of device if the user is not concerned with which one of several similar devices is assigned. SYSDEV, in turn, selects the appropriate channel manager (of managers if the device has more than one channel), informs it of the ID of the new owner of the device, and passes to the user the process ID of the channel manager(s). Until the owner releases the device, other process may not communicate with it.

BEfore I/O can begin, the user must communicate an open request to the device channel manager. This establishes a path between the user and the device handler to which is sent all subsequent access and transfer requests. The user is passed the ID of a general purpose interface process which receives all I/0 requests and passes each to the correspondinding device handler for action.

A write access request is used to specify the type and amount of data for the handler to expect. In return, the user is informed by the nandler the status of the device and the process.ID to be used for data transfer. A buffer is prepared in the interactive services subsystem **252** to receive data from the user.

A read access request instructs the handler to retrieve data from a device for the user by loading a buffer. In response, the handler provides a status report on the read operation, operation complete, error conditions, etc.

A put request follows a write access and initiates data transfer from the user's buffer to the interactive services subsystem **252** butter and eventually to the device itself. The response from the handler provides the header (process ID) to be used during data transfer to the handler.

A get request follows a read access, includes the header (process ID) to be used for data transfer, and intiates transfer by the handler from the buffer (loaded from the device by the read access) to the user.

It is possible to open two devices, prepare one for read and one for write, and then effect transfer of data from one to the other independent of any transfers to, or from the controlling user process. A transfer request can reference any combination of devices and datasets.

This clears a current read, write, get, put, or transfer, and releases buffers for further use. The user returns to the open condition with respect to the device.

The communications path, set up with the open command between the user and the handler, is broken. Any subsequent attempt to access the same handler would be rejected.

When use is no longer required of a device, the user can return it to the system by releasing it back to SYSDEV. It is possible that another job requires the device and has had its request queued awaiting release. In this case, the device will be immediately reassigned to the queued job.

Ownership of a device is via the job number of the owning job, not the process ID of the process that acquired it. This signifies that if a job consists of a network of processes (all with the same job number), then once a device has been acquired by a job, any other process in the same job can also communicate with the device provided it "knows" the appropriate process ID to be used. This is especially useful in allowing backup processes to take over devices when a primary process fails.

The Pascal interface to these steps is via the following matching set of procedures and corresponding functions:

VACQNAME—Acquire from SYSDFV by name (ID), by type, and subtype.

VACQLIST—Acquire from SYSDEV by name (ID), by type, and subtype.

VOPEN—Open communications path.

VREADDEV—Access request.

VREADCRT—Access request.

VWRITEDEV—Access request.

VGET—Data transfer (disk or tape datasets).

VPUT—Data transfer (disk or trape datasets).
VTRANSFER—Data transfer (disk or tape datasets).
VENDIO—Terminate access request.
VCLOSE—Close communications path.
VRLSEDEVICE—Release back to SYSDEV.
VCONTROL—Send control data.

C. Supported Devices

To support a peripheral device on the interactive services subsystem requires two components—a hardware channel controller to physically interface the peripheral's data and control channels to the interactive services subsystem **252**, and a device channel handler process in the interactive services subsystem **252** to logically interface the device's channels with the formal I/O protocol described in the previous section. Available controlled include a serial channel controller **254**, **255** that allows devices with asynchronous serial duplex RS232 data channels to be attached, and a magnetic tape controller **714** that interfaces the data and control channels of magnetic tape drives. Other channel controllers may be used for synchronous and X.35 channels.

Dumb terminals are simple, byte-at-a-time VDU/-keyboard devices used in the system **100** via the interactive services subsystems **252** as program development system terminals. Attachment to the interactive services subsystem **252** is through an asynchronous 9.6Kbps duplex line to a serial channel controller **254**. Pressing a previously defined log-on key on an idle dumb terminal results in the initiation of a program development system in a virtual machine which then acquires the terminal to solicit control and interactive communication with the terminal user.

A data message sent to the terminal from an application consists of a variable length string of ASCII control and data bytes, which may write to the screen or perform control functions in the terminal, depending on the individual characters sent. The simplest message is a single character sent to echo a keyboard input. The terminal handler provides an acknowledgement to the user for each message sent to the handler.

Characters from the keyboard are sent individually and directly to the owning process as they are received by the handler. The terminal handler is given the process ID of its owner when acquired and sends each typed character to the owning process in an individual unsolicited notification message. To the user process, the arrival is an unsolicited event which can be waited or checked on. The status of the terminal, as seen by the interactive services subsystem **252** can be verified at any time by the control function available to the user to detect if the terminal is on-line, off-line or suspended by the system operator. To reduce the burden of the application having to echo each character, a local handler level echo option can be chosen.

To send messages to the terminal, standard I/O routines and protocol are used with open, write access, put, end I/O, and close as meaningful for this device. To maintain integrity, the handler times out if gaps in a message string occurs with the timeout value related to the band rate of the serial channel. The read access and get functions have no meaning for this device.

Intelligent terminals **270** are used as consoles for system programmers, system operators, and operator stations. The intelligent terminal is a microprocessor-controlled device which maintains a number of "virtual screens" on the one physical display by partitioning the screen into a number of separately managed areas. Each visible partition is used to provide human interface to a different class of functions allowing for efficient and flexible use of the one device. Non-visible partitions serve as buffers and simplify implementation of categorical lists of operator-selectable functions not actually displayed.

In a telephone answering support system application, this terminal is used as part of a composite device called an operator station **106**. This consists of an intelligent terminal **266** augmented with a voice headset **224**, separately connected via an audio channel to the real-time subsystem **230**.

When the telephone answering support system application attempts to acquire an operator station **106**, SYSDEV must jointly allocate the data channel to the terminal and the audio channel to the headset. However, the data channel to the terminal and the voice channel to the headset are independently controlled and operated once they have been acquired and the intelligent terminal, used by the telephone answering support system in the operator station **106**, is identical to that used for system consoles in so far as this description is concerned.

Messages sent to an intelligent terminal are similar to those sent ot a dumb terminal, but only to the extent that they concist of a variable length string of bytes. However, intelligent terminal messages have a type-attribute also which is the start-of-message character indicating to the terminal how to interpret the characters that follows. Additionally an end-of-message character is defined, to delineate strings of messages. Messages are used to define message partitions; enable or disable keybaords and function keys; define screen partitions; read, write, and move data in the terminal's buffer; and many other functions.

Messages from the terminal can be either unsolicited resulting from operator keyboard actions, or solicited by a request to read the contents of a partition on the screen. The latter is useful in data-entry environments, where a partition can be set up to contain a data entry form consisting of protected labels and unprotected response fields, with the operator allowed to enter keyboard data directly into the terminal in a local mode. During data-entry to the terminal there is no workload on the system, which can then quickly scan the whole form with a single read access, get I/O sequence. This results from an unsolicited stimulus from the terminal caused by actuation of the send function key.

Before the system permits log-on, the intelligent terminal solicits a personnel ID, password, and function selection code from the user. The log-on information collected is transferred by the handler via an unsolicited notification packet to SYSDLO, which receives all unsolicited inputs from unowned devices. SYSDLO uses the function selection code to effect initiation of a corresponding interactive log-on process. The latter process verifies the identity of the physical terminal, user, and password to assess whether the user will be allowed to proceed or not.

All of the I/O functions can be used with the intelligent terminal with control and timeouts applying as for the dumb terminal.

Printers are connected to the interactive services subsystem **252** via serial asynchronous channels terminating on a serial channel controller **254**, **255**. The software printer handler in the interactive services subsystem **252** can support any number of printers and in a

normal configuration, at least two are attached each to a different serial channel controller **254, 255** for security. The system monitor (SYSMON) acquires printers to produce selective hard-copy of operator interactions and dumps of the system log. To avoid long-term suspension of processes requiring printers when printers have been acquired by SYSMON (and therefore unusable by any other process, even if idel), a print spooler job owns the system printers rather than any individual system or application job. As a result, a user generally sends data to be printed to the publically writeable print spooler dataset, where it is eventually output to a free system printer. In addition, other printers may be attached to the system which are available for general allocation to any job requesting acquisition of a printer for private use. The protocol between a user and printer handler follows the standard sequence outlined earlier, but with read access and get requests being illogical for this particular device.

The printers supported by the interactive services subsystem **252** are the Printronix models 300/600, which have a wide range of advanced print options including variable print spacing, variable line spacing, variable character size, a programmable vertical forms unit and plot mode.

The printer handler manages the attributes of the printer on behalf of the user, who can select the required print characteristics and operational mode using the control functions provided in the I/O protocol.

There are three basic operational modes which the user can adopt when sending output to the printer.

Transpartne Transmission Mode—All characters received by the printer handler are passed to the printer with no code translation or insertion of control codes by the handler. This allows the knowledgeable user complete control of the printer's functions.

Edited Transmission Mode—The user sends the printer handler complete message, with the handler managing the insertion of carriage control characters to produce a correctly-structured printout, as defined by the user in previous control messages.

Line-Oriented Transmission Mode—Data is sent to the printer handler on a line-by-line basis with the first word of the data interpreted by the handler as a format control word. This allows attributes of the printout to be changed on a line basis, rather than on a message basis as occurs with the edit transmission mode, although the facilities offered to both modes are similar.

The printer is equipped with a soft vertical form unit which the user can program via control messages to skpply any desired pattern of vertical carriage movements in response to vertical tab control characters. This greatly simplifies printing out highly structured items such as bills or invoices.

The printer can also be set to plot mode in which the bit-wise contents of the data sent to the printer are printed in a corresponding dot pattern in a left-to-right, top-to-bottom scan. This can be useful in printing super-large character sets, defined in software, for printout labeling. Another application of plot mode is the output of information and statistics in graphical formats such as histograms, charts and graphs.

Datasets critical to both speed of access and data transfer rate are stored on disk drives with each drive or group of drives controlled by an individual disk data processor **934, 936** in the file services subsystem **908**. For off-line archive storage or large datasets with low access priority, magnetic tape is a cost-effective media. The low data rates associated with magnetic tape permit the interactive services subsystem **252** to host the magnetic tape device handler, in addition to the other device handlers. Each magnetic tape controller **714** attached to the interactive services bus **704, 708** can drive two tape drives **718, 720** with up to seven separate magnetic tape controllers permitted per interactive bus **704, 708**.

User processes do not directly acquire and use magnetic tape drives as they would a printer. Users are concerned with the datasets supported on the magnetic tape, not in the device as such. The magnetic tape datasets are of the sequential unmapped type with variable length records. To the user, there is no difference whether a disk-based dataset or one contained on a magnetic tape volume is used. Either way the create-open-access-transfer-use protocol is used.

However, in the event a user attempts to open a dataset on magnetic tape, SYSDIR acquires a magnetic tape drive from SYSDEV and ensures that the correct tape reel containing the dataset is mounted. SYSDIR then responds to the user's open request, supplying the interactive services subsystem **252** process ID to provide access to the magnetic tape device handler for all subsequent read/write access and transfer requests. To the user, there is no difference in handling the magnetic tape device handler in an interactive services executive **702, 706** or the disk handler in a disk data processor **934, 936**. Each is identified via a similar process ID to which the same requests can be sent.

The tape volumes used conform to ANSI X-3.27, Level 1. This provides for a standard label on a tape header allowing for transportation of tapes between a Delta and other processing systems. Level 1 support limits each tape to one dataset (that is, no directory is supported on the tape), but a single dataset can span across multiple tape volumes. Support for the more sophisticated ANSI X-3.27 levels (Levels 2–4) will be available in the future as needs dictate.

The magnetic tape device handler supports writing and reading of sequential records, the reading of any portion of a record, and a maximum record size of 4096 bytes. The handler also permits the user to send control messages to the tape drive allowing rewind and erase to be activated under control of the user process.

The real-time subsystem **230** switches and processes continuous voice traffic, interfacing to the system **100** as analogue signals at the telephone room subsystem **206, 214, 216**. Each voice channel is carried on a separate pair of copper wires attached at a remote point to switching equipment through which the voice signals are routed to the system **100**. This equipment could be a common carrier's exchange plant, or a private organization's internal PBX. In certain telephone answering support system applications, the voice lines originate as off premises extensions attached in parallel to the subscriber's own telephone line at the main distribution frame of the telephone company's exchange (secretarial lines).

Two factors must be taken into account in relationship to normal telephone applications: (1) the average utilization of each line is low, and (2) the cost of lines from a central office to a remote site is high.

The remote line concentrator **202** allows a large number of low utilization remote lines to be selectively switched onto a smaller number of high utilization trunks leading to the system **100**. The concentrator is located in, or adjacent to, the exchange equipment and

connected via the central office main distribution frame to each subscriber line for which service is required. The number of trunk lines necessary between the concentrator and telephone room subsystem **206** is related to both the service levels required, and the traffic patterns that occur. Although variable depending on the nature of the services offered and the particular business environment being serviced, a concentration ratio of 60:1 would be typical in a standard telephone service system application.

For control, the concentrator is connected via redundant low speed serial data links **210** to the interactive services subsystem **252**. The data link **210** carries messages in both directions, including:

(1) Detection of rings, status changes and acknowledgment of action requests, together with the identity of the lines or trunks on which the activity is taking place.

(2) Requests to seize a selected subscriber's line and connect it to one of the voice trunks; to release a subscriber's line and disconnect it from the trunk; and to post alarm conditions intended to invoke local service responses from system **100** to the concentrator.

Since the concentrator control lines are attached to serial channels in the interactive services subsystem **252**, the concentrator handler is resident in the interactive services subsystem **252** and the concentrator is considered tn interactive services subsystem peripheral device.

The concentrator **202** has a fault-tolerant hardware architecture and each contains two controllers with either controller capable of operating as primary and the other as backup. Five voice trunks and one data channel terminate on a concentrator trunk board. Each trunk board is operated by the primary controller and is installed in pairs. Any single subscriber line can be switched to any of ten trunk positions of a pair of trunk boards. From two to ten real trunks can be installed on a pair of trunk boards, and the pair is treated by the concentrator handler in the interactive services subsystem **252** as a single logical concentrator. However, a single physical concentrator rack at the remote site can contain up to four logical concentrators (four pairs of trunk boards, each paid with two data links and up to ten trunks). Subscriber lines are partitioned into ten groups, and a group can be connected to have access to only one logical concentrator trunk set. In situations of higher traffice density, this allows up to 40 trunks to be configured between one physical concentrator and the system **100**. The datasets **208, 212** and link **210** can thus be physically implemented as up to four parallel sets.

Since the concentrator is remote from the interactive services subsystem **252**, dataset modems **208, 212** are required at each end of each serial data control line **210**. However, since the data rate on the line is low (110 or 300 Baud), simple low cost modems are sufficient.

Although subscribers are terminated on the exchange side of the concentrator, individual subscriber lines are still managed by the system as though they were attached to the telephone room subsystem **206** even though routed through the concentrator over common trunks. A user can send SYSDEV an acquire request for a subscriber line using the concentrator terminated channel ID to represent the normal telephone number of the ringing subscriber line. SYSDEV then requests the concentrator handler to seize the requested line, select a free trunk, and switch the line onto the selected trunk. The concentrator handler passes to SYSDEV the channel ID of the incoming common trunk onto which it has switched the requested line. SYSDEV then sends the real-time subsystem **230** a request to assign the trunk to the original requestor so that it can become the owner of the trunk and perform switching or processing functions with it.

The concentrator handler manages the assignment of trunks from the concentrator, not SYSDEV. These trunks are a resource used by the concentrator handler in fulfilling requests to switch a line to telephone room subsystem **206**. SYSDEV manages the ownership of the actual subscriber lines in the remote exchange location as if they were physically terminated on the telephone room subsystem **206**. Once a user has acquired and seized a line via a concentrator, the channel ID of the trunk (employed by the concentrator handler in connecting the remote line to the real-time subsystem **230**) is used in switching and processing functions, not the actual telephone number of the ringing telephone lines.

Once a line has been acquired via SYSDEV, the only interaction between a user and the concentrator handler is control functions. At any time, the owner can send a status request to the concentrator handler to measure the current state of the remote line. Additionally, a user can acquire a remote line without seizing it, as described previously, with the actual seize performed at a later time. In this case, the seize command is sent in a control message to the concentrator handler, and the trunk assignment is then returned to the owner-user. When a user is finished with the remote line, a release request is sent to SYSDEV which informs the concentrator handler to free the subscriber line at the concentrator and return the trunk used back to the pool of available trunks. SYSDEV also sends a message to the real-time subsystem supervisor informing it that the released trunk is no longer owned by the user.

D. Interactive Services Subsystem Hardware Architecture

An interactive services subsystem **252** consists of three hardware components, as shown in FIG. 7.

Interactive services executives **702, 706** are one of the five types of processors used in a system **100** with each subsystem containing two interactive services executives **702, 706**. During normal operation, the workload involved in peripheral I/O is shared between the two interactive services executive processors **702, 706**. In the event of a failure of one interactive services executive **702, 706** or the bus **704, 708** it controls, the survivor takes over the full load until the failed member is replaced.

Interactive buses **704, 708** are each controlled by one of the interactive services executive processors **702, 706**. The interactive buses **704, 708** known as the U-Bus and the V-Bus respectively. Each bus has sixteen parallel data lines, a parity line, and an associated set of control lines. Data can be transferred along the bus between attached device controllers and the interactive services executive's data memory as follows:

(1) Write—3.478 Mbytes/second.

(2) Read—2.857 Mbytes/second.

(3) Instantaneous—7.5 Mbytes/second.

Each U and V-Bus pair **704, 708** can have attached up to 64 device controllers. Each controller is dual-ported bo both the U and V-Bus, contains internal data buffers and microprogrammed control logic for all channels, and interfaces to device-specific I/O channels.

Requests for I/O services arrive at an interactive services executive **702, 706** as packets on the inter-

processor executive bus **912**. **914**, where they are passed by the interactive services executive's resident executive to the appropriate process. To accomplish an I/O transfer, handler processes load a buffer area in the data memory of interactive services executive **702, 706** with control or data information, and then request the bus control hardware of the interactive services executive **702, 706** to transfer the contents to a specific device controllers' I/O or control channel.

Alternatively, the data from a device controller channel can be requested to be loaded into a data memory buffer using the same microprogrammed DMA hardware that is part of the U and V-Bus control extension unit **726** of an interactive services executive **702, 706** and a device controller are via the particular interactive bus **704, 708** to which that interactive services executive is attached. Data transfers across the bus are asynchronous, with the actual data movement on the channels leading from the controller to the peripheral devices; thus each device controller contains extensive buffering to provide temporary storage for each channel. Device controllers can be accessed from either the U or V-Bus **704, 708**, allowing the devices attached to be accessed should one interactive services executive **702, 706** or its associated bus fail.

An interactive services executive **702, 706** is formed by adding, to a standard processor module **724**, an interactive processor extension unit **720** which enhances the basic 16-bit standard processor module **724** processor in the following areas as shown in FIG. **8**:

(1) Adds extension **808** to the high-speed program memory of the standard processor module to increase capacity from 12K words up to 32K, of which 8K are the extensions **812** to manage the hardware additions to the standard processor module **724**.

(2) Adds 64K words of data memory **806** primarily used to define data buffers for passing blocks of information between a user and a device controllers' channel through the interactive services executive **702, 706**.

(3) Extends the basic instruction set of the standard processor module by over 30 instructions used by the interactive services executive-resident programs to test and control the hardware additions.

(4) Adds a microprogrammed interactive bus controller **802** which manages, asynchronously to the standard processor module, the transfer of data between buffers in the interactive services executive's data memory **806** and device channel buffers in the device controllers. The controller has two methods of communication with the standard processor module. The first is a 64-word deep interrupt FIFO buffer **804** used to pass to the standard processor module the data memory addresses of full (on input) or empty (on output) data buffers when the microcontroller has completed a requested transfer. The other is a 4K word block of high-speed memory, occupying the top of the standard processor module's data memory address space, containing a list of device controllers to be polled plus a word count/date memory address pair for each buffer associated with the input and output of each device channel for each controller. To effect a transfer, a device handler loads the origin of the buffer and the word count. As the controller for the device channel is polled if the associated channel buffer has input or can accept output data, the bus controller effects the transfer of the data, decrements the word count, and increments the data memory buffer address. When the word count reaches zero, the bus controller posts the final buffer address to the interrupt FIFO, thereby notifying the standard processor module that the transfer is complete.

The serial channel controller **254, 255** is a device controller allowing up to 16 RS232C serial, duplex, asynchronous channels to be connected to the interactive services subsystem **252**. Each channel's data rate can be independently selected from between 110 baud to 38.4K baud, and each channel has the following active signals:

serial transmitted data,
serial received data,
data terminal ready, and
ready to send.

A 32-byte input and output buffer is associated with each of the 16 channels and data is transferred between these channel buffers and the universal asynchronout receivers/transmitters that physically drive each channel. Data desinted for or received from a serial channel is passed between these 32-byte buffers and the buffers in the interactive services executive data memory, under control of the microprogram in the interactive services executive bus controller **802**. Each channel has associated with a four-bit control register which configures the channel for input, output, or both and assigns the channel to either the U or V-Bus **704, 708** and consequently either the U or V-interactive services executive **702, 706**. A process in the interactive services executive **702, 706** sets the contents of these registers via a device controller by writing a command word into any of the top 16 locations in the interactive services executive data memory. The microcontroller in the interactive services executive bus controller **802** detects an attempt to write to these memory loations and transfer the written command word across the U or V-Bus **704, 708** to the particular channel control register, as specified in the low 10 bits of the command. Other device controllers use the same technique to configure their I/O channels, allowing the device controller to respond correctly when the interactive services executive bus controller **802** polls it for input data or attempts to transfer data for output.

The magnetic tape controller **714** is a device controller which interfaces with magnetic tape units through a formatter **716**. It contains an 8K word buffer that decouples data transfers between the magnetic tape controller **714** and the magnetic tape on one side, and the interactive services executive **702, 706** on the other. The magnetic tape controller **714** uses a microprogrammed controller to manage the interface to the drives, communicate with the interactive bus controller in the interactive services executive **702, 706**, and control accesses to the magnetic tape controller **714** internal buffer and control registers.

A magnetic tape controller **714** has three channels through which a handler process in the interactive services executive **702, 706** communicates:

(1) A control channel used to pass control data to the magnetic tape controller **714**. This channel allows buffers in the magnetic tape controller's 8K memory to be defined by loading various magnetic tape controller control registers. One particular control word loads a register which activates the physical control lines to the magnetic tape unit, enabling the magnetic tape handler process in the interactive services executive **702, 706** to select a drive; set read and write thresholds in the formatter; rewind the tape; erase; write file marks, and other similar tape unit control functions.

(2) A bidirectional data channel, along which data can be transferred between a buffer in the interactive services executive's data memory and the magnetic tape controller's buffer. Once data has been transferred to the magnetic tape controller's buffer, it is passed to the selected drive by the magnetic tape controller independent of any activity on the U or V-Bus **704, 708** between the magnetic tape controller **714** and the interactive services executive **702, 706**. The magnetic tape controller **714** can read or write data between a tape unit and one area of its 8K memory while the interactive services executive **702, 706** is simultaneously reading or writing to another area in the same buffer memory. This double-buffering allows data transfer operations to be either pipelined to a single drive, or interleaved between drives.

(3) An interrupt channel extends from the magnetic tape controller **714** back to the interactive services executive. Within the magnetic tape controller **714**, a one word buffer is used to store the results of a read or write operation to the tape unit **718, 720**, containing the termination status of the transfer and the number of data bytes actually transferred. The interrupt channel is used by the magnetic tape device handler to read these results back to the interactive services executive **702, 706**. Filling the buffer causes the buffer identity to be stored via the DMA termination FIFO **804**, thus resulting in an interactive services executive **702, 706** interrupt.

The three channels to a magnetic tape controller **714** are implemented in the same way as any other channel used to communicate with a device controller: a buffer to send or receive channel data is defined in the interactive services executive data memory, and a description of the buffer is put in the polling list of the interactive services executive bus controller **802**, which initiates the requested transfer between the buffer in data memory and the magnetic tape controller **714**.

An interactive services executive process can also communicate directly with the magnetic tape controller **714** by writing a command word to any of the top 16 locations of the interactive services executive's data memory. This command word is self-addressed in the lower six bits as to which device controller it is desinted for, and is transferred immediately by the interactive services executive bus controller **802** to the destination magnetic tape board (for example, one of up to 64 controllers on the U and V-Bus **704, 708**). In the case of the magnetic tape controller **714**, this command word can request the status of the magnetic tape controller's internal registers, assign a magnetic tape controller to either the U or V-Bus **704, 708**, clear the magnetic tape controller **714** to power-up state, and enable or disable the control channel in the magnetic tape controller **714**.

Data passed to the magnetic tape controller **714** along the control channel from the handler process is used not only to control the tape drive units, but to define how the magnetic tape controller's 8K memory is to be partitioned. Buffers of up to 4K words size can be specified. The size of the data blocks transferred can be chosen to relate to the size of the dataset's logical records contained on the magnetic tape.

Data is transferred between the buffers in the interactive services executives' data memory **806** and the channel buffers in the device controllers under control of the microprogrammed U and V-Bus controller **802** in the interactive services executive **702, 706**. Transfers take place one word at a time on a polling basis, rather than as continuous streams of data from one buffer to one channel. Therefore, if a number of data transfers are pending at the same time, all are effectively undertaken in parallel by interleaving individual one word transfers as determined by the sequence in the polling list. This uniformly distributes among all channels the effects of all traffic, thus minimizing the queueing problems that would occur if one channel could occupy the bus solely during transfer of a large block of data.

The transfer sequence is as follows: the microcontroller accesses an entry in the polling list which specifies the individual device controller to be polled on the U or V-Bus **704, 708** depending on which the interactive services executive **702, 706** controls. The controller is polled by the interactive services executive bus control microcontroller **802** and responds with the ID of the next channel for which a data transfer is required or possible together with the direction of transfer. One word of data is transferred between the interactive services executive's data memory buffer indicated in the polling list entry and the specified channel buffer in the device controller. The bus controller then continues to the next device controller specified in the polling list.

If more than one channel on a device controller is active, data is then transferred from each active channel in a round-robin sequence, with one data transfer occurring on each poll cycle. The advantage of this technique is that inactive channels, or those logically removed from the system, take no overhead in the polling process since they are either missing from the list or do not respond to the bus controller's poll. The polling sequence of device controllers is entirely programmable, since the polling list forms part of the data memory space of the interactive services executive processor **702, 706**. It is also possible to poll a given controller at a greater frequency by inserting its address more than once in the polling list. This allows for dynamic alterations to be made in the servicing of specific device channels.

E. Interactive Services Subsystem Software Architecture

A number of processes are created during the logical initialization of an interactive services executive **702, 706** and divided into two basic categories. The first consists of processes which are not related to any one specific device type bus which provide general management and interface functions. The second encompasses the device handlers, each controlling all the devices of a given type which have been assigned to the host interactive services executive **702, 706**.

Management and interface functions include the following processes:

(1) Device management maintains tables which relate devices to their owning job numbers, responds to assign requests from SYSDEV, and communicates with the other interactive services executive of the interactive services subsystem pair **702, 706** for load sharing and backup purposes. Device owners open and close their devices through this function.

(2) Input/output interface receives all requests to access and transfer data from any device. This function interfaces the request to the appropriate device handler.

(3) Buffer managers manage the use of the interactive services executive's data memory for buffered data transfers between the interactive services executive **702, 706** and the inter-processor executive bus **912, 914**, and also between the interactive services executive **702, 706** and the device controllers.

(4) Backup/recovery shadows operations in the other interactive services executive of the subsystem and manages takeover in case of a device or bus failure.

When a job needs a device, it sends an acquire request to SYSDEV either specifying a unique device, or requesting any one of a given class of similar devices. SYSDEV identifies one that satisfies the request and sends an assign request to the interactive services subsystem **252** shown as having the device attached. There could be up to four interactive services subsystems **252** and within each, devices are load-shared between the U-interactive services executive **702** and the V-interactive services executive **706** according to a predefined plan dictated to the interactive services subsystem **252** by the system configuration file.

The assign request is sent to the device management function of the primary interactive services executive of the subsystem. The primary interactive services executive is either the one which first completed initialization when the subsystem was powered-up, or alternatively the one which survived an interactive services subsystem failure, becoming and afterwards remaining primary by default.

Primary interactive services executive then determines which of the two interactive services executives **702, 706** is responsible for this request, marks the ownership details in its tables, and passes the request to backup interactive services executive to maintain its tables. The response to SYSDEV includes the process ID of one of the two device management functions, the primary or that in the backup where the owner is to send subsequent requests. This process ID is passed to the requesting process by SYSDEV. The requesting process, or any other process in the same job to which it passes the process ID, can then send requests directly to the interactive services executive **702, 706** with no need to involve any higher-level kernel system functions.

At this stage, the owner can send to device management an open request that checks the status of the device and also puts the user in touch with the interactive services executive's input/output service function. The process ID of this process is returned in response to the open request, and used in all subsequent accesses to the opened device.

The owner can then send to I/O services read and write access requests. Each such request is acknowledged back to the owner with an access PID, which is used in subsequent get and put requests to cause the physical transfer of data between the device's buffers and the owner. A user can build up a number of interleaved transfers by using multiple Read/Write access requests to the same device separated by End I/O (ENDIO) requests after each access. The I/O service function transforms these user level requests into internal calls to the specific device handler, which then performs the physical data transfer to and from the device.

Transfers are terminated by sending an End I/O request to the I/O service function. The device channel is closed with a request to the device management function. Closing the channel invalidates the reference value supplied in the open request response, and any subsequent attempt to call I/O service after a close results in a negative response.

When an owner no longer needs a device, a release request is sent to SYSDEV which deallocates the device in its tables and informs interactive services subsystem device management with a free device request.

The owner can then send to I/O services read and write access requests. Each such request is acknowledged back to the owner with an access PID, which is used in subsequent get and put requests to cause the physical transfer of data between the device's buffers and the owner. A user can build up a number of interleaved transfers by using multiple Read/Write access requests to the same device separated by END I/O (ENDIO) requests after each access. The I/O service function transforms these user level requests into internal calls to the specific device handler, which then performs the physical data transfer to and from the device.

Transfers are terminated by sending an END I/O request to the I/O service function. The device channel is closed with a request to the device management function. Closing the channel invalidates the reference value supplied in the open request response, and any subsequent attempt to call I/O service after a close results in a negative response.

Then an owner no longer needs a device, a release request is sent to SYSDEV which deallocates the device in its tables and informs interactive services subsystem device management with a free device request.

If a failure occurs while an interactive services executive **702, 706** is performing I/O with a device, the data involved in that particular get (input) or put (output) transfer is lost since the data buffers in the failed interactive services executive **702, 706** are no longer available. However, because the device controllers are dual-ported, they can still be accessed from the surviving interactive services executive and I/O restarted by re-initiating the interrupted data transfer request, get or put.

To the owner, this failure is detected either as a time-out to an access or transfer request, or a transfer which completes in error. The owning process must then re-initiated the interrupted get or put. If a transfer was not in process, then action need not be taken by the user.

The next open request will be automatically rerouted to the surviving device manager process using the system bus ID mechanism in the interprocessor communications network.

If a device controller fails, all devices connected to that controller are lost unless the device itself is redundantly connected to the interactive services subsystem **702, 706** via two sets of I/O channels terminating on two different device controllers. This occurs in the case of the concentrator which has internally redundant channels for high availability. Each logical concentrator has two serial data channels connecting it to two different serial channel controllers **254, 255** on the interactive services subsystem **254**. Each logical concentrator manages up to ten voice trunks to the telephone room subsystem **206, 214, 216**.

SERVICE SYSTEM ACCESS RESPONSE

As illustrated in the telephone service system flow chart shown in FIG. **12**, the service system **100** responds to an incoming call by first identifying the type of line upon which the call has been placed and then seizing the line when specified answer conditions have been met. For most types of lines, the answer conditions will specify immediate answering of the call. However, for secretarial lines the client may specify that the line be answered only after a selected number of rings, and the number of rings may vary with time of day and day of week. For example, the client may have a business

telephone connected as a secretarial line and wish to have his own secretary answer the line during normal business hours. Therefore he may specify that during his regular business hours the phone will be answered by the system **100** only when his secretary is unable to answer the phone within five rings. Outside of normal business hours he may wish to have the phone answered immediately.

In any event, the line is seized upon satisfaction of the indicated answer conditions and further response then depends upon the type of line involved, the dictates of the caller, and the options selected by the client.

For example, in the case of a secretarial line the client can specify either an operator assisted answering service or a fully automatic answering system service. In the event that operator assisted answering has been selected by the client, upon seizure of a secretarial line, the system **100** identifies and connects to the line an available operator station. Client account data is automatically and immediately sent to the visual display unit at the operator console to assist the operator in responding to the call. This data may include the salutation with which the call is to be answered, instructions for order taking, messages for selected callers, information about the client's schedule or where he can be reached and so forth. Upon terminating the call the line is released and the operator station becomes available for the next operator assisted call.

In the event that the client has selected automatic answering service, the caller is greeted with a client selected greeting and invited to leave a message at the occurrence of a tone. The client may record his own greeting and change it at will, or alternatively, may use a system provided greeting which does not specifically identify the called client. Furthermore, the caller may specify the length of any message which can be recorded and the maximum number of messages which may be stored by the system. In any event, upon generation of the tone, the voice service system receives and records in the inbasket portion of the client message basket any message dictated by the caller. Up to the maximum message time specified by the client. During this recording process the system responds to message editing commands as if the caller were a system client. However, to avoid confusing nonclient callers, no editing prompts are provided and an unsophisticated caller may simply dictate a nonedited message with no knowledge of the system editing feature. Upon receipt of the message, the call is terminated and the line is released.

A direct incall line, like a secretarial line, is associated with a particular message basket. However, unlike the secretarial line, the direct incall line is not available for general use by the client and is dedicated to the receipt of incoming calls for the message basket. Upon seizing a direct incall line, the system proceeds to the automatic answering mode in a manner which is essentially the same as the automatic answering response to a call on a secretarial line.

A general incall line is similar to a direct incall line except that a general incall line is not associated with a particular message basket. Therefore, upon seizing a general incall line the caller is prompted to enter a message basket code. Upon entering the required message basket code, the line becomes dedicated to a particular client message basket and the response proceeds in the same manner as the automatic response to a secretarial line or a direct incall line. Execution of a change command enables a system user to specify a new message basket code at any time and return to the automatic answering step with the line dedicated to a new message basket. This feature enables a caller to access the system on a general incall line and leave messages for several different message baskets with a single call and without having to redial the line for each different message.

A general access line provides all of the functions available on a general incall line and also affords a system user access to count ownership functions. Upon seizing a general access line, the caller is prompted to leave either a message basket code or a personal ID code. If the client leaves a message basket code the system associates the line with the indicated message basket and the call is answered in the automatic answering mode similar to the response to a general incall line.

On the other hand, if the caller enters a personal ID code the caller is granted access to the account ownership functions such as retrieval of messages stored in his inbasket, sending of messages through his outbasket to other inbaskets or telephone numbers, changing of the client greeting and so forth. Through the change function, the caller may at any time change states to leave messages with other selected message baskets or perform other ownership account functions without redialing the call.

A direct recall line is associated with a particular message basket and provides access to only account ownership functions for that message basket. Such a line may be utilized to implement a high security environment by requiring the personal ID code to have a first field which is established when the account is open and a second field which can be changed at will by the caller as an account ownership function. Such an arrangement requires a person accessing the account to know the direct recall line telephone number, the first field of the personal ID code and the second field of the personal ID code. Such an arrangement is thus relatively secure and precludes access by a person entering personal ID codes at random in the hope of accessing an interesting account.

The general recall line is similar to the direct recall line except that it is not dedicated to a particular message basket. The caller is prompted to enter a message basket code and then must enter a personal ID code corresponding to the selected message basket code. This gives the caller access to the account ownership functions.

Referring now to FIG. **13**, most user initiated commands which are entered through the DTMF tone signals of a telephone keyboard begin with the asterisk key which interrupts any current activity by initiating a pause and serves as an attention key for the system **100**. Virtually any state can be interrupted with a new command sequence although some command sequences will not be valid for certain states. For example, a caller accessing the system **100** through a direct incall line can never exercise account ownership functions.

System **100** responds to the pause command by interrupting any current function such as storage or retrieval and initiating a five second time window during which the system **100** will respond to additional commands. If a further command is not entered within this time window, the system will remain in a pause state and will not respond to additional commands until the asterisk key has been reactuated. The use of the window created by the asterisk key precludes the unintentional generation of command signals which might affect the state of the system. For example, it is known that the voices of some

people have tonal characteristics which cause the voice to generate the dual tone signals corresponding to certain keys. Unless the asterisk key has been actuated, these signals are simply ignored by the system **100**. At the same time, once a user actuates the asterisk key and places the system in a pause mode, it is unlikely that the user will be speaking to the system or otherwise creating sounds that might be misinterpreted as a keyboard actuation.

Once the user has actuated the asterisk key a specific command may be entered. For example, immediate actuation of the number sign key effects a sign off or termination of activities with respect to the current account. This in effect returns the system to a state in which the user is prompted to enter a message basket number to associate the call with a particular account. In the case of a secretarial line or direct incall line which is associated with a single answering function for a dedicated message basket, the call would simply be terminated.

Actuation of the one key institutes a listen command which enables the user to retrieve messages recorded in the user's inbasket by listening to the playback of previously recorded voice messages. Again, this function would be unavailable for certain states such as for calls received on a secretarial line or a direct or general incall line. The listen key can also be used to listen to a dictated message for editing purposes.

Actuation of key **3** creates a talk command which causes the system **100** to receive and record voice messages. This command would be appropriate for any recording mode such as normal telephone answering mode, recording messages in a client's outbasket for forwarding to other inbaskets or telephone numbers, or recording a client's personal greeting, depending upon the particular recording responsive mod the system **100** is in at the time the talk command is executed.

The **4**, **5** and **6** keys provide convenient editing functions. The **4** key causes the recording system to back up five seconds while the **6** key causes the system to go forward by five seconds. The number **5** key is a continuation key which in effect terminates the pause initiated by the asterisk key and continues the system **100** to the state which preceded actuation of the asterisk key.

The **7** and **9** keys provide clear and save functions which may be utilized to preserve or erase user messages stored in an inbasket during a retrieval mode or to edit messages stored in a client's outbasket for delivery elsewhere. Key **8** provides an insert function which enables a voice message to be inserted or deleted between other parts of a message during the recording of a message. For example, were the user to record a message and then wish to insert a sentence between two preceding sentences the user could actuate * **4** for a number of times until the recording point is backed up in five second increments to a point preceding the insertion point. Upon actuating * listen (**1**) the user may listen to the prerecorded message until the insertion point is reached. At this point the user actuates * insert (**8**) and may begin dictating the inserted portion of the voice message. Upon completion of the insert, the user merely actuates * listen (**1**) to listen to the remaining portion of the previously recorded message until the end thereof is reached. Alternatively, the user could use the * forward (**6**) command to rapidly skip forward in the previously recorded message to the end.

The insert function may also be used to mark the beginning and the end of a voice message segment. The segment can then be deleted with the * clear command.

Actuation of the zero key initiates a help function in which detailed voice message prompts are communicated to the system user to explain the current state of the system and to explain which keyboard combinations should be actuated to obtain a desired objective.

In the event that the system is in a state in which a personal ID code number should be entered, the user is prompted to do so by actuating the asterisk key, the appropriate number keys, and the pound sign key (enter). The enter key is utilized to terminate all number sequences but the system will attempt to validate a previously entered number sequence after a selected (five second) timeout even in the absence of actuation of the enter key.

Actuation of the **2** key creates a change function which must be further defined by actuation of a third key stroke. For example, actuation of the **1** or listen key places the system in a retrieval mode with the system **100** responding by beginning to retrieve and communicate to the user any voice messages stored in the corresponding inbasket

Actuation of the **2** key initiates a change account function which enables the system user to enter a new message basket code number terminated by the enter key and thereby gain access to the inbasket of a different message basket. This combination of commands enables a caller to leave a voice message in several different inbaskets with a single call.

Actuation of the number **3** talk key places the user in a recording mode for recording of a message in the user's outbasket (assuming that this mode is authorized). Actuation of the **5** key creates an administrative mode in which the caller may utilize additional function commands to execute selected account ownership administrative functions. For example, actuation of the star **1** or star **3** keys would enable the user to listen to or change the client greeting. Other key combinations may be assigned in the client administration mode to enable selection of various client account options such as the maximum length of an inbasket message, the maximum number of inbasket messages, specification of the second field of a direct recall line personal ID code and so forth.

Execution of the * **2** zero key sequence causes an active operator station to be connected to the line to give the user access to an operator for assistance in executing any command sequence that the user might be having difficulty with or for execution of any account administration commands that might require operator intervention.

While most command functions are initiated with the asterisk key the entry of certain code number sequences such as a message basket number, delivery code, a distribution list, or a telephone number are not preceded by the asterisk key. It will be noted that these entries are in the nature of a data specification and not strictly a command function.

While FIGS. **12** and **13** describe the functional operation of system **100** from the user point of view, FIGS. **14–26**, to which reference is now made, describes the functional operation of system **100** in terms of the response of the system **100** itself.

To customize voice message service to the needs of a particular client, the system **100** utilizes client service options and parameters. Client service options make a

particular voice messaging features available or unavailable to a client while client service parameters are quantifiers that may be varied within a specified range to suit a client's need (e.g., the number of messages that may be left in a message basket).

To provide these capabilities, voice messaging service is organized into the functions shown in FIG. **14**, the voice messaging service functional flow block diagram. The first function, obtain call information, provides the voice messaging service access to the client instructions pertaining to delivery of a message or information concerning the phone line over which the call has been received.

Depending upon the call information, voice messaging performs either the message delivery function or the select account/activity function. As part of the select account/activity function, client information including client service options and parameters is obtained. The principal task of this function, however, is the invocation of the function providing the particular service desired by the client. The select account/activity function determines which specific service is desired based on the call information, information entered by the client or commands entered by a voice messaging service information operator on behalf of the client.

The answer call function, which is shown in greater detail in FIG. **16**, plays out a message prepared by the client, usually a salutation. Depending on the client information, the voice message service will then record a single message from the call which may or may not employ the recording control primitive commands, except a single command from the caller again invoking the select account/activity function or perform the terminate call function.

The send messages function is illustrated in greater detail in FIG. **18** and places messages and delivery instructions in the client's outbasket. The retrieved messages function which is illustrated in greater detail in FIG. **20**, enables a client to examine the client's message basket. The change administrative data function is shown in greater detail in FIG. **25** and enables a client to change selected call and administrative instructions associated with the client's account. The client may selectively and repeatedly exercise these functions during a call. The message delivery function is illustrated in greater detail in FIG. **26** and enables a client's message to be delivered to the inbasket of any client or any telephone. The attempt to deliver the message commences at the time specified in the delivery instructions and, for telephone delivery, continues in accordance with those instructions.

The terminate call function is the last function performed and handles disconnection of the call and bookkeeping of the activities performed during the call.

The interrupt driven keyboard functions may be commanded by a client at any time and hence are not explicitly shown in FIG. **14**. These functions include help, operator assistance, editing, and the change function which is illustrated in FIG. **14** and returns the system operating state to the select account/activity state.

Also not shown in FIG. **14** is a maintain usage data function. This function is embedded throughout the flow process and collects and makes available to the system **100** the data describing the resources used by each call for client billing and system operating purposes.

Extensive prompts are available to a system user to explain use of the system. Any command that can be entered by a client except request operator can be entered by a service system information operator. Each command is preceded by the asterisk symbol. If no command is entered by a user in response to a prompt, the prompt is not repeated and the call is directed to a system operator to provide assistance to the caller. The operator then remains connected until the call is terminated or until the operator executes an end assistance command through the operator terminal. If a user attempts to enter an inappropriate command, an error prompt is communicated to the user which explains the mistake and provides the user with an opportunity to retry the command. After a number of retries specified by a system parameter, error retries, the user is provided with a help prompt. If a correct entry has not been received after the help prompt has been received repeated once, the caller is either disconnected or referred to an operator, depending upon a selected client service option.

The call information needed to control a voice message service operation consists of either line type or delivery information. Line type is provided for calls received at the voice message service facility. Delivery information is provided if message delivery is to be performed. A direct line is an exclusive line associated with one and only one client and is used solely to receive calls. The system **100** does not attempt to use a direct line for telephone delivery and does not attempt dial out on a direct line. A service schedule is always associated with a direct line. A general line is associated with no particular client and may be used for telephone delivery. Delivery information is derived from delivery instructions prepared by the client. It identifies the outbasket holding message and the particular delivery instruction being executed.

Making reference to FIG. **15**, the select account/activity function provides the client access to specific voice messaging services. The function is invoked by one of the following events:

1. A change in function caused by entry of an access send messages, an access retrieve messages, an access administrative or access account client command.
2. Receipt of a call over a shared line.
3. A request operator client command.
4. Receipt of a call over a direct line.

In the first case an analyzed command function determines if additional information is needed. If so, a prompt requesting the needed information is provided and the await caller entry function is invoked. Once the caller has completed his entry, the validity check caller entry function is invoked. If the entry is valid the obtain client information function is invoked. If it is not valid either a prompt is provided to the caller or if repeated attempts at personal identification number code entry have occurred, the call is referred to an operator. Once required client information is available the obtain client information function is invoked to determine if the requested service is available to the client. The function providing the service is invoked if the service is available and the service not available prompt is provided if it is not.

In the event of a shared line call, the initial client prompt is provided. If no DTMF entry is received within the time specified by a system parameter, prompt interval, after the prompt is provided, the operator assistance function is invoked. In the normal situation, a

DTMF entry is made and the await caller entry function is invoked and the flow proceeds as in the first case. If an operator has been requested the operator assistance function is invoked.

If a call is received on a direct line, the client information is uniquely associated with a line and the obtain client information function is invoked immediately. If the client information specifies that operator assisted call answering is required the operator assistance function is removed. Otherwise the answer call function is invoked.

The analyze command function shown in FIG. **15** is invoked as the result of an access send messages, and access retrieve messages, an access administrative function, or an access account client command. The access account command cannot be executed until the message basket number of personal identification number code is provided. The access send messages, access retrieve messages, and access administrative functions commands cannot be executed unless the client has entered his personal identification number code. The analyze command function shall determine the data needed, if any, and provides the request identifier message basket prompt to the client.

For the access send messages, access retrieve messages, and access administrative functions commands, no prompt is provided if the client has previously provided his personal identification number code. The obtain client information function is then immediately invoked. Once a prompt has been provided, the await caller entry function is invoked and operates to accept the entry of data by the caller.

If the data entered by the caller uses data syntax, it is assumed to be the message basket number. If it is in command syntax, it is assumed to be the personal identification number. On a shared line allowing entry of partial message basket numbers, (a portion of the number being uniquely associated with the incoming line), the data entered by the client is assumed to be low order digits of the message basket number and is concatenated with the base number to form the message basket number.

The validity check entry function verifies that the data entered designates a valid message basket or personal identification number code. If it does not, the invalid message basket or invalid personal identification number code prompt is provided depending on the syntax of the data entered (with or without asterisk key). If the third attempt at personal identification number code entry is found to be invalid, operator assistance function is invoked. No attempt is made to corrolate the type of data entry with the type requested by the prompt. No attempt is made to cross-corrolate personal identification number and codes and message basket numbers. If the entry is valid, the obtain client information function is invoked.

The obtain client information function accesses and acts upon client information uniquely associated with the line over which a call is received, with the message basket number entered by the client, or with the personal identification number code entered by the client. Client information is created using the customer support system, which is interactive software system used to maintain the data base directing voice message service system operation, to report system performance, and to support administrative activities. The client information includes a client identifier, client service options, the client message basket number, and the client personal identification number code IPIN). The client service options include:

1. Call answering available.
2. Autocall answering provided.
3. Operator assisted call answering provided.
4. Edit controls available (CA).
5. Send messages available.
6. Delivery codes employed.
7. Distribution lists employed.
8. Edit controls available (SM).
9. Retrieve messages available.
10. Message basket status provided.
11. Inbasket review provided.
12. Output basket review provided.
13. Reply/redirect services provided.
14. Message amendment provided.
15. Delivery instructions amendment provided.
16. Telephone delivery available (auto).
17. Telephone delivery available (operator assisted).
18. Auto delivery reply.
19. Operator assisted delivery reply.
20. Reply edit controls available.
21. Call answer enter disconnect employed.
22. Administrative functions available.
23. Multiple salutations employed.
24. Call forwarding available.
25. Message basket forwarding available.
26. Distribution list modification available.
27. Default delivery option.
28. Shared account used.
29. Partial message basket number used.
30. Time of delivery prompt form.

The client service parameters are:

1. Maximum answer message.
2. Maximum message size.
3. Maximum number of messages.
4. Maximum number of addresses.
5. Number of distribution lists.
6. Number of delivery codes.
7. Base message basket number.
8. Number of salutations used.

Delivery instructions consist of addressee information and a delivery code. When placing a message in his outbasket, the client may specify the addressee directly by providing a message basket number or telephone number, or he may specify the addressee indirectly by providing a distribution list number. A client distribution list consists of a set of predefined addresses and delivery codes prepared using the customer support system (CSS).

Each client is invited to preestablish a specified number of single digit delivery codes. This number is specified by the client parameter number of client codes. The voice message service system maintains additional single digit system defined delivery codes which may also be used by the client. The following information is provided for each delivery code:

1. Delivery code number.
2. Time of first attempt and time to stop trying. The time in the delivery code shall be interprinted in terms of the local time at the voice messaging service system installation.
3. Retry interval.
4. Total number of attempts.
5. Automatic telephone delivery.
6. Ready to receive acknowledgement not required.
7. Called party identification required.
8. Delivery acknowledgement required.

9. Called party response permitted.

The principal action of the obtain client information function is the comparison of requested activity with the client service options to determine if the request can be honored. If it cannot, the service not available prompt is provided and the await caller entry function is invoked. If the request can be honored, the function invoked depends upon the event originally invoking the select account/activity function as follows:

1. If invoked by the access send messages, access retrieve messages, or access administrative functions command, the respective function shall be invoked. The client information controlling this action is that associated with the personal identification number code.

2. If invoked by the access account command, the operator assistance or the answer call function is invoked in accordance with the client information associated with the match to each basket specified.

3. If invoked by the receipt of a call over a shared line and if a message basket was entered by the caller, the operator assistance or answer call function shall be invoked in accordance with the line information associated with the message basket specified.

4. If invoked by the receipt of a call over a shared line and a personal identification number was entered by the caller, the default function specified in the client information associated with the personal identification number code shall be invoked.

5. If invoked by receipt of a call over a direct line, the operator assist or answer call function is invoked in accordance with the client information associated with the line.

The answer call function shown in FIG. **14** is described in greater detail in FIG. **16**, to which further reference is now made. This function enables callers to directly access the message receiving facilities of a client. The prompts and subordinate functions employed by the voice messaging service system answer call function are illustrated. The operator assisted answer call function is considered part of the operator assistance function. Depending on the client service option, the select account/activity function (FIG. **14**) invokes one or the other of these two functions when a call is received over an exclusive line or when a valid change account command is honored.

The answer call function always begins with playout of a salutation selected by the client. If the caller is determined to have gone on-hook during the playout, the terminate call function is invoked. If the playout is completed and the client service option does not allow a caller reply, the terminate call function is invoked.

If the client service option allows a caller reply, a second client service option determines if the caller can leave only a simple telephone answering machine type of reply or if the full range of voice message service edit controls are availabe to him. In the former case, the record invitation prompt is provided and the record message function is invoked. In the latter case, the record/edit invitation prompt is provided and the edit message function is invoked. However, in this case, if no DTMF tone is detected within the time interval specified by the system parameter prompt interval, invocation of the edit message function is cancelled and the record message function is instead invoked. If a prolonged period of silence elapses during recording, the record message function will provide the disconnect warning prompt and may invoke either the terminate call or operator assistance function in accordance with the client service option. At the option of the client, the record/edit invitation prompt is omitted and superseded by the record invitation prompt even if message editing has been selected as a client service option.

The duration of the caller reply is controlled by the client parameter maximum answer message. If the caller message exceeds the value of this parameter, the time exceeded prompt is provided and the terminate call function is invoked. If the reply is completed within this time, the complementary close prompt is provided and the terminate call function is invoked.

Depending on a client's service option, the change account command may be recognized within the answer call function. If this client service option is in effect, a voice message service system **100** delays invocation of the terminate call function for the period of time specified by the system parameter change account delay. If the option is in effect, the command is recognized within the edit message function and during the period of time that the final prompt is being played. It is not recognized during playout of the salutation or initial prompts within the record message function, or once the terminate call function is invoked.

The playout salutation function provides playout of a salutation selected by the client. If multiple salutations are available, the salutation select shall depend upon the time of day and day of the week upon which the call is received and the salutation selected shall be the one whose service schedule matches the time and date. If the caller goes on-hook during the playout, the terminate call function is invoked.

The record function simply records data received over the telephone line for no more than a specified period of time. This period of time is specified by the client parameter maximum answer message. No DTMF tones are recognized by this function. The message is placed directly in the client's inbasket and hence, if the caller exceeds the specified period of time, this function will retain that portion of the message that had been received in the client's message basket.

If a period of silence exceeding the length specified by the system parameter silence interval elapses while recording, the system **100** shall provide the caller with the disconnect warning prompt. The caller may override the imminent disconnection by immediately resuming recording. The voice message service system will either disconnect the call or invoke the operator assistance function in accordance with the client service option when the prompt interval expires if recording is not resumed.

The edit messages function is illustrated in greater detail in FIG. **17** and provides the client or caller with complete voice message editing facilities. Upon invocation, this function records the signals received over the telephone line and simultaneously monitors these signals for DTMF tones representing client commands as shown by the record and enter command functions. Command entry causes one or more of the subsidiary functions shown or one of the keyboard functions to be exercised. If the edit message function has been invoked by the answer call function, only the access account, help and abort options within the interrupt function are available. The commands invoking the other interrupt functions are ignored. The commands invoking the various voice message service system edit controls are shown to the left of the flow line leading to the subsidiary function.

The pause command is embedded in every client command as its first DTMF character. Hence, the stop record/playback functon always performs as any part of any command entry. The record function is invoked as the result of a talk command. The oversize message prompt is issued by this function as a message exceeds the client parameter maximum message size. Under certain conditions, the save command terminates operation of the edit message function. Otherwise, all other commands eventually result in the invocation of the enter command and/or playback function. The action of the insert command depends upon whether it is the first, second, or subsequent entry of this command during a particular client activity. If it is the first entry, the mark segment beginning function is invoked. If it is the second entry, the mark segment end function is invoked. Subsequent entries are ignored. The action of the clear command depends upon the current state of the insert commands. If one command is in effect, the delete segment function is invoked and then the enter command and record functions are invoked. If two insert commands are in effect, the delete segment function is also invoked. In this case, however, the enter command and playback functions are invoked following the deletion. If no insert is in effect, the delete message function is invoked. In any case, no insert command is in effect when the action is completed.

The action of the safe command also depends upon both the current state of the insert command and whether or not anything has actually been recorded since the first insert command was entered. If an insert is in effect and anything has been recorded, the save segment function is invoked. If an insert is in effect and nothing has been recorded, the delete marks function is invoked and if no insert command is in effect, the save message function is invoked. In all cases, no insert command is in effect when the action is completed. Usage data collected within this function includes net amount of storage used.

The record function directs recording of the non-DTMF signal received over the voice channel associated with the call. DTMF tones are not recorded. On each invocation of this function, a tone prompt is provided to indicate that recording is now in progress. The enter command function directs the monitoring of the signal received over voice channel associated with the call for DTMF tones. If a tone is detected, it is analyzed for the DTMF characters received, and if a client command has been entered, it invokes the function appropriate to the command.

The playback function directs playback of the message over the voice channel associated with the call. The stop record/playback function suspends the action of the record or playback functions. The mark segment beginning function notes the current position of the message as a segment beginning. The mark segment end function notes and marks the current position in the message as a segment end.

The position playback function positions the messages for playback in accordance with the command causing invocation. If invoked by the backup command, it positions the message backwards by a factor equivalent to the system parameter positioning precision or to the beginning of the message if application of the factor so dictates. If invoked by the forward command, it positions the message forward by a factor equivalent to the system parameter positioning precision or to the end of the message if application of the factor so dictates. If invoked by the listen command, it positions the message to its beginning. After positioning the message, the playback function is invoked.

The delete segment function deletes the portion of the message between the segment beginning and segment end. If segment end has not been specified, it deletes the portion of the message following the segment beginning. It deletes the notations of segment beginning and segment end. If, after the deletion, the segment end is not also the end of the message, it positions the message to the point immediately following the end of the deleted segment. If the segment end is also the end of the message, it positions the message to a point immediately following the end of the deleted segment.

The delete message function deletes the entirety of the message so far recorded.

The save segment function saves the segment of the message just recorded (i.e., the portion of the message recorded since the first insert command was recognized). The segment is logically entered into the message at the point noted as the segment beginning. If a segment end is in effect, the function deletes the portion of the original message between the segment beginning and segment end. Upon playback, the message, including the segment, is heard as a single continuous message. At the completion of this function either the segment beginning or segment end exist (i.e., the effect of the insert commands will have been cancelled).

The delete marks function deletes the notations of segment beginning and of segment end.

The save message function preserves the message recorded, if any, for disposition by the function that originally invoked the edit message function.

The send message function which is indicated in FIG. **14** and illustrated in greater detail in FIG. **18**, provides clients with the capability of recording and sending one or more messages to clients or nonclients at a single session. Upon invocation, the send message function examines the client outbasket. If the total number of messages in the message basket (i.e., the sum of the message in the inbasket plus those in the outbasket) exceeds the client service parameter maximum number of messages, it provides the message basket full prompt and immediately invokes the terminate send messages function. (The client must employ the retrieve messages function to create room in the his message basket.) However, the room created will not be available until the client enters a signoff or a valid access account command or until he hangs up.

If the message basket is not full, the send messages introduction prompt is provided and the edit message function is invoked. The client prepares his message using this function. When the client signifies to the edit message function that the message is complete, the send messages function provides the delivery instruction invitation prompt.

The client may then begin entry of the delivery instruction, in which case the accept delivery instruction function is invoked. Alternatively, the client may elect to defer providing delivery instructions by entering the save command. In either case the next instruction invoked is the place message in outbasket function. After the message has been placed in the outbasket either the command invitation prompt or message basket full prompt is provided. The message basket full prompt is provided if the number of messages currently in the message basket equals or exceeds the client service parameter maximum number of messages. If the mes-

sage basket full prompt is provided, the terminate send messages function is necessarily invoked. If the command invitation prompt is provided, the client may elect to send another message (by entering the top command), or he may terminate the message sending activity (by entering any of the change function commands or the signoff command or by hanging up). In the latter case, the terminate send messages function is invoked.

The accept/edit delivery instructions function shown in FIG. **18** is illustrated in greater detail in FIG. **19**. A delivery instruction consists of a delivery address and a delivery code. The set of delivery instructions associated with the message is a set of delivery address/delivery code pairs ordered by the sequence in which they were entered by the client with the delivery address always preceding the delivery code. In the absence of the accept/edit delivery instructions function, an element is either member of the pair (i.e., a delivery address or a delivery code). An element is entered by the client using the data syntax. To enable the client to enter and verify his delivery instructions, the voice messaging service system enters into a dialog with him. It accepts an input from him, plays out a prompt or element, and accepts an instruction from him. This instruction may direct playout of another element, delete the element display, or may be the next element. When an element has been played out, the next instruction also determines the disposition of the element just played.

The accept/edit delivery instructions function has basically four subsidiary functions which are: client entry, validity check client entry, save element, and delete element. Depending on the client instruction that caused invocation of the save element function, this function may act upon the just played element, the current element or the previous element in the ordered set of delivery instructions. The echo-back element prompt may apply to the just entered, the next, the previous, or the current element. In FIG. **19** this fact is indicated by the parenthetical comment included along with the function representation.

The client entry function accepts entry in both command and data syntax. The commands may cause a prompt to be provided, one or more of the subsidiary functions to be exercised, or one of the change function, help, abnormal terminal request, or operator (CHAO) functions to be exercised. The commands associated with a particular flow line are shown to the left of the line.

The information comprising a delivery instruction is entered using data syntax and such an entry causes both the validity check client entry and save element functions to be invoked. As a result of its operation the validity check client entry function provides the echo-back element prompt, the invalid element prompt, or the next element prompt. The save element function monitors the number of delivery instructions entered. As a result of its operation, it may cause the delivery instructions capacity prompt to be provided.

The various commands entered by the client permit the client to step through the set of delivery instructions, delete elements, and terminate operation of the function. Instruction entered by the client, the save element, function may or may not be invoked.

The forward command causes either the echo-back (next) element prompt to be provided and save (previous) element function to be invoked or all the elements played prompt to be provided. In enables the client to step through the set of elements in a forward direction.

The backup command causes either the echo-back (previous) element prompt to be provided and the save (current) element function to be invoked or has the same effect as the listen command. It enables the client to step through the set of elements in a backward direction.

The listen command causes the first element to be provided and then the echo-back (first) element prompt to be provided. It places the client at the beginning of the set of elements. The continue command causes the echo-back (current) element prompt to be provided. The clear command causes the delete element function to be invoked if an element has just been echoed. The command has no effect otherwise. The save command invokes the save element function if needed and terminates operation of the accept delivery instruction function.

The client entry function directs monitoring of the signal received over the voice channel associated with the call for DTMF tones. If a tone is detected the function determines if the entry is employing data syntax or command syntax. When the entry is complete and if the entry is in need of syntax, the function invokes the validity check client entry in save previous functions.

On a shared line allowing entry of partial message basket numbers, the data entered by the client is assumed to be low order digits of the message basket number and is concatenated with the base number to form the message basket number. If the entry is in command syntax and if it requests one of the CHAO functions, the requested function is invoked.

If the command is forward and there is no next element, the all elements play prompt is invoked. If there is a next element it is echoed. If the command is in response to the echo of an element, the save element function is invoked from the previously echoed element. If the command is not in response to the echo of an element, the save element function is not invoked.

If the command is backup and if there is at least one delivery address in the set of delivery instructions, and if there is no prior element, the first element prompt is provided followed by the echo-back element prompt for the first element in the set of delivery instructions.

If the set of delivery instructions is empty, the backup end is ignored. If there is a prior element it is echoed. If the command is not in response to the echo of an element, there can be no further action and the client entry function is again invoked. If the command is in response to the echo of an element, the save element function is invoked for the previously echoed element. Note that echo of the delivery code precedes the delivery address. If the command is listen and the set of delivery instructions is empty, the listen command is ignored. If the command is listen and if there is at least one delivery address in the set of delivery instructions, the first element prompt is provided followed by the echo-back element prompt for the first element in the set of delivery instructions. If the command is continue and if the command is in response to the echo of an element, the echo-back element prompt is provided using the same element. Otherwise the continue command is ignored.

If the command is clear and if the command is in response to echo of an element, the delete element function is invoked for that element, otherwise, the clear command is ignored.

If the command is save and if the command is in response to the echo of an element, the save element function is invoked for that element. Operation of the

accept delivery instructions function is then terminated. All other commands are ignored.

The validity check client entry function shown in FIG. **19** recognizes an entry in data syntax as consisting of a delivery code, a message basket, a telephone number or a distribution list in accordance with the number of digits in the item. Where one digit represents a delivery code, two digits represent a distribution list, three to nine digits represent a message basket number, and ten or more digits represent a telephone number. If the entry is not valid, the invalid element prompt is provided and the client entry function is invoked.

If the entry is valid, it is echoed back to the client and the client entry function is invoked. The save element function is invoked as previously indicated.

If special instructions (those not covered by standard addresses or preestablished delivery codes, e.g., delivery may be made to either John Doe or Mary Smith) are required on an address, then the client obtains a voice message service information operator by entering the request operator command. The save element function is invoked by the client entry function if the entry is in response to the echo-back element prompt and if the entry is not a clear command. This function saves the delivery instructions and monitors the number of addresses to which the message is being sent. In all cases the function saves an element that was echoed back. In certain cases the function saves an additional element or a delivery code as follows:

1. If the client service option delivery codes employed is not in effect, the function also saves a delivery code element for the address just echoed. The delivery code value is immediate.

2. If the client service option delivery code employed is in effect and if the element echoed back was not a delivery code, the function saves a delivery code element for the address just echoed. The delivery code value corresponds to the value established by the client service option default delivery option.

If a distribution list identified was echoed back, the set of delivery addresses and delivery codes comprising the distribution list is explicitly saved. Upon replay of the set of delivery instructions associated with the message, each delivery instruction in the distribution list is explicitly echoed and the distribution list identifier cannot be retrieved. If a number of addresses exceeds the client parameter maximum number of addresses, no further delivery instructions can be saved and the delivery instruction capacity prompt is provided.

The delete element function deletes any element just played back to the client.

The place message in outbasket function illustrated in FIG. **18** is invoked subsequent to the accept delivery instructions function and places the message in the client outbasket and monitors the number of messages in the message basket.

The originator (sending client) retains ownership of a message until it has been received by all addressees. Until received, the originator may cancel the message, edit it, or change delivery instructions using the retrieved messages function if appropriate client service options are in effect.

The terminate send messages function schedules message delivery. If a message is being delivered to a client, it appears in the client's inbasket at the time specified in the delivery instruction. If a message is being delivered to a telephone number, the telephone delivery function is invoked (message delivery FIG. **14**) at the time specified in the delivery instruction.

The terminate send messages function prepares a summary of client send messages activity including:

1. Client identification.
2. Message basket identification.
3. Time activity began.
4. Time activity ended.
5. Process time used.
6. Change in disk storage used.

The terminate and send messages function also prepares a detailed record of client retrieved message activity on a message by message basis that includes:

1. Client identification.
2. Message basket identification.
3. Message identification.
4. Time playout began.
5. Time playout ended.
6. Dispostion of message and special services employed (e.g., cleared, saved, replied, amended, redirected, operator assistance functions employed, etc.).
7. Processor time used.
8. Disk accesses used.
9. Change in disk storage used.

The retrieve messages function shown in FIG. **14** is illustrated in greater detail in FIG. **20**. This function provides the capability for a subscriber to retrieve messages within his message basket. When a client enters his PIN code number, voice message service system **100** recognizes the caller as a client and invokes the retrieve messages function. The retrieve messages function begins with the introduce retrieve messages prompt. Following this prompt invocation of the playout message basket status, review inbasket, and review outbasket functions is dependent upon a client's service option. A separate option controls the availability of each of these functions.

Invocation of review inbasket and review outbasket functions is also dependent upon the presence of messages in the inbasket or outbasket. If no messages are present, the function is not invoked. In this case, if the playout message basket status function has not been invoked, the inbasket empty or outbasket empty prompt respectively is provided.

The playout message basket status function begins with a message basket status prompt. This prompt reports the number of received messages, if any, available for retrieval by a client and the number of messages, if any, awaiting delivery, or the fact that the entire message basket is empty. The review inbasket function is shown in greater detail in FIG. **21** and enables the client to listen to the messages in his inbasket. If there are no messages in the inbasket, this function has not been invoked. Consequently, it shall commence playout of the messages and simultaneously await entry of a client command. Client commands invoking the CHAO functions, the delete message function, and the retain message function are always effective if the review inbasket function can be invoked. The commands invoking the remaining functions are effective if the appropriate client service functions are in effect.

Upon completion of message playout, delete message, retain message, or the valid reply redirect sequence, voice message service system **100** plays out the next message in the inbasket, or if all messages have been reviewed, the inbasket review complete prompt. Messages are played in reverse order of receipt (i.e., the last

message received is the first message played out). The playout message status function always repositions the current message back to the beginning. The reply function is not invoked if there is no room for reply in the client's message basket. Instead, the message basket full prompt is provided. The await client command function shown in FIG. **21** directs the monitoring of the signal received over the voice channels associated with a call for DTMF tones. If a tone is detected and if the tone is in command syntax, this function suspends playout of the message. If the command requests one of the CHAO functions, the request function is invoked. If the command is continue, playout of the message is reinstated at the point it was suspended. If the command is clear, the delete message function is invoked. If the command is save, the retain message is invoked. If the command is listen and the appropriate client service option is in effect, the provide message status function is invoked. If the command is talk and the client service option allowing reply/redirect is in effect, the reply/redirect function is invoked. If this option is not in effect, the talk command is ignored. All other commands are ignored although the playout is suspended. The playout message function shown in FIG. **21** directs playout of a message over the voice channel associated with a call.

The provide message status function provides the time of receipt prompt, repositions the message to its beginning, and reinstates playout of the message after the time of receipt prompt has been provided. The form of the time of receipt prompt subject to a client service option and may be either precise (e.g., message received at 1:30 p.m. on the 6th of June) or general (e.g., message received this morning). At the time the message was delivered, the time of receipt (TOR) was associated with the message.

If the general form of this prompt is used, it shall use the time at which this in-progress call was begun to compute the length of time the message has been in the inbasket. The precision of this computation shall be one-quarter hour. This duration, D, and the time of receipt (TOR), are, shall be used to select the general time of receipt (GTOR) prompt.

The delete message prompt shown in FIG. **21** deletes the entirety of the message and invokes the delete acknowledgement prompt.

The retain message function preserves the recorded message for subsequent use of the client and provides the save acknowledgement prompt. It notes that the message has been played out.

The reply and redirect functions provide the capability to record a commentary to a received message. Depending on a client instructions, the commentary is returned to the originator of the message as reply, or redirected to a set of addresses specified by the client, with or without the original message.

The reply functional flow is illustrated in greater detail in FIG. **22** which shows the prompts and subsidiary functions employed by the reply/redirect functions.

These functions are not invoked unless the appropriate client service option is in effect and there is room in the client's message basket for the reply. Upon invocation, the function provides the reply location prompt, invokes the form copy function to place a copy of the client's inbasket message into his outbasket, and invokes the await instruction to obtain the response to the reply location prompt.

The response to this prompt indicates that the reply is to preface the original message, to replace the original message, to be appended to the original message or that the message is merely to be redirected. In the first three cases the message function shall be invoked and in the fourth case accept edit delivery instructions function shall be invoked.

Upon normal completion of the edit message function, the concatenate messages function is invoked if the response to the reply location prompt indicated that the reply was to preface or be appended to the original message. The reply disposition function is then invoked. Client instructions within the reply disposition function may effectively redirect the reply. In this case, the set delivery instruction function is invoked. The reply function concludes by invoking the original disposition function.

The await instruction of FIG. **22** directs monitoring of the signal received over the voice channels associated with a call for DTMF tones. If a tone is detected, the function analyzes the DTMF characters received, and if the client command has been entered, the function invokes the function appropriate to the command. If a command invoking one of the CHAO functions is detected, the requested function is invoked. If a talk command is detected, it is interpreted as an indication that the reply is to be appended to the original message. If a save command is detected, it is interpreted as an indication that no reply is planned and the message is merely being redirected. If a clear command is detected, it is interpreted as an indication that the reply is to replace the original. If an insert command is detected, it is interpreted as an indication that the reply is to preface the original message. All other commands are ignored. The form copy function places a logical copy of the current inbasket message in the outbasket. The edit message function has been previously described. The message being edited is the current outbasket message which was just created by the form copy function.

The concatenate messages function concatenates the current inbasket message with the current outbasket message and places the result in the outbasket. If the response to the reply location prompt was the insert command, the outbasket message precedes the inbasket message. If the response to the reply location prompt was the save command, the inbasket message precedes the outbasket message.

The reply disposition function provides the reply disposition prompt. This prompt identifies the address of the sender of the current inbasket message. In response to this prompt, a command invoking one of the CHAO functions, a save command, a clear command, or an entry in data syntax may be received. If a command invoking one of the CHAO functions is received, the requested function is invoked. If a save command is received, the delivery instruction for the reply consists of the address of the sender of the current inbasket message with a delivery code of immediate. The original disposition function is invoked. If a clear command is received, the sender of the original message is not included in the delivery instructions (i.e., no delivery instructions are now associated with the reply), and the edit/accept delivery instructions function is invoked. If an entry in data syntax is received, the delivery instruction associated with the reply is the same as that provided if the save command had been entered. However, the edit/accept delivery instructions function is invoked before the original disposition function.

The client may redirect a received message entering delivery instructions for the message. This function provides the addressee invitation prompt and invokes the edit/accept delivery instruction function.

The original disposition function provides the original disposition prompt. In this response to this prompt a command invoking one of the CHAO functions, a save command or a clear command may be received and acted upon. All other commands are ignored. If a save command is received, retain message function is invoked. If a clear message is received, the delete message function is invoked.

Returning to FIG. **20**, the review outbasket function enables the client to listen to the messages in his outbasket and, especially, the delivery instruction associated with them. It also enables him to complete sending, editing, or addressing of a message whose composition was interrupted. This function is illustrated in greater detail in FIG. **23** to which reference is now made.

If there are no messages in the outbasket the review outbasket function has not been invoked. Consequently, this function commences playout of the messages and simultaneously awaits entry of a client command. Client commands invoking the CHAO functions, the delete message function, and the retain outbasket message function are always effective if the review outbasket function can be invoked. The commands invoking the remaining functions are effective if the appropriate client service options are in effect.

Upon completion of the message playout, delete message, retain message outbasket message, or the valid amendment sequence, the voice message service system **100** either plays out the next message in the outbasket or, if all messages have been reviewed, the no more messages prompt. Messages are played in reverse order of entry. The provide delivery status function always repositions the current message to the end of the message and the beginning of the delivery instructions.

The await outbasket command function directs monitoring of the signal received over the voice channels associated with the call for DTMF tones. If a tone is detected and if the tone is in command syntax, this function suspends playout of the message. If the command requests one of the CHAO functions, the requested function is invoked. If the command is continued, playout of the message is reinstated at the point it was suspended. If the command is clear, the delete message function is invoked. If the command is save, the retain message function is invoked. If the command is listen and the appropriate client service option is in effect, the provide delivery status function is invoked. If the command is talk and the client service option allowing amendment is in effect, the amend message/instructions function is invoked. If the option is not in effect the talk command is ignored. All other commands are ignored although the playout is suspended.

The playout of the message function is followed by playout of the delivery instructions.

The provide delivery status function skips playout of the current message and provides the delivery status prompt. This prompt reports the status of each message awaiting delivery or for which delivery acknowledgement was specified. The delivery status of a message is reported as pending, unaddressed, delivered or undelivered. Additional status information is provided as follows:

1. For pending and undeliverable status in the event of message basket delivery, date/time of availability to the addressee for message retrieval. For pending an undeliverable status in the event of telephone delivery, date/time of most recent attempt and reason (busy, no answer, etc. of other reason as determined by the voice message service system information operator on assisted delivery attempt).

2. For delivered status the delivery date and time. This status exists only for messages with delivery acknowledgement specified in delivery instructions. Until all addressees have received the message, the sender can obtain delivery status from the voice message service information operator.

3. For unaddressed status, none.

The delete message function has been previously discussed.

The skip to delivery function causes a skipping of the playout of the current message and commences playout of its associated delivery instruction. The message remains in the outbasket.

The amend message function shown in FIG. **23** is illustrated in greater detail in FIG. **24**. This function provides the capability of amending a message and is not invoked unless the appropriate client service option is in effect. Upon invocation it provides the type of amendment prompt, and waits to obtain the response to this prompt.

Response to this prompt will indicate that the amendment is to preface the current message, to be appended to the current message, to replace the current message or that only addresses are to be amended. In the first two cases, if room is available in the outbasket, the form outbasket copy function is invoked. If room is not available, the outbasket full prompt is provided. If the copy is formed and always in the third case, the edit message function is invoked. In the fourth case, the amended delivery instruction function is invoked if the client service option allows and ignored if it does not.

Upon normal completion of the edit message function, the concatenate messages function is invoked if the response to the type of amendment prompt indicated that the reply was to preface or to be appended to the original message. If the client service option allows, the amendment delivery instructions function is then invoked.

If in response to the type of amendment prompt a command invoking one of the CHAO functions is detected, the request function is invoked. If a talk command is detected, it is interpreted as an indication that the amendment is to be appended to the original message. If a save command is detected, it is interpreted as an indication that only delivery instruction amendment is planned. If a clear command is detected, it is interpreted as an indication that the amendment is to replace the original. If an insert command is detected, it is interpreted as an indication that the amendment is to preface the original message. All other commands are ignored.

The form outbasket copy function places a logical copy of the current outbasket message in the outbasket (i.e., two copies of the message are now in the outbasket).

The edit message function has been explained previously.

The concatenate outbasket messages function concentrates the original message with the original message and places the result in the outbasket. If the response to the type of amendment prompt was the insert command, the amended message precedes the original message. If the response to the type of amendment prompt

was the save command, the original message follows the amended message.

The amend instructions function shown in FIG. **23** provides the amendment instructions prompt. In response to this prompt, a command invoking one of the CHAO functions, a list command, a save command, a clear command, or a entry in data syntax may be received. If a command invoking one of the CHAO functions is received, the request function is invoked.

If a listen command is received, the accept delivery instructions function is invoked and playout of the delivery instructions, if any, or the no delivery instruction prompt will occur as described for that function.

If a save command is received the delivery instructions associated with the original message are also associated with the amended message. Playout of the next outbasket message or the no more messages prompt will then occur. If an entry in data syntax is received, the delivery instructions associated with the original message are also associated with the amended message and the accept/edit delivery instructions function is invoked. All other commands are ignored.

The terminate message retrieval function shown in FIG. **20** is invoked to schedule or cancel message delivery as appropriate if an addressed reply to an inbasket message, an amended and addressed outbasket message or amended delivery instructions to an outbasket message exist. This function prepares a summary of client retrieved messages activity including:

1. Client identification.
2. Message basket identification.
3. Time activity began.
4. Time activity ended.
5. Processor time used.
6. Change in disk storage used.

This function prepares a detailed record of client retrieve message activity on a message-by-message basis that includes:

1. Client identification.
2. Message basket identification.
3. Message identification.
4. Time playout began.
5. Time playout ended.
6. Disposition of message and special service employed (e.g., cleared, saved, replied, amended, redirected, operator assistance functions employed, etc.).
7. Processor time used.
8. Disk accesses used.
9. Change in disk storage used.

The change administrative data function shown in FIG. **14** is illustrated in greater detail in FIG. **25**. This function gives the client control over certain of the administrative and control data associated with his account. This function is invoked by the change administrative data command (*, 2, 5). In response to this command, the administrative menu prompt is provided. Selections from the menu use data syntax, not command syntax (i.e., the selection is not preceded by the asterisk key). The majority of the selections, as the menu shows, are subject to client service options.

The edit salutation function provides a client the capability of recording and altering the telephone answering voice salutation. This function is subject to a client service option. A client service option allows multiple salutations. If this option is in effect, the select salutation function is invoked to obtain identification of the particular salutation to be edited. The client is able to record a new salutation in a manner similar to recording a message as described for the send message function. The principal distinction is that no delivery instructions are required.

Upon completion of a new salutation, the system **100** replaces the current salutation with the newly recorded one. The maximum length of the salutation is specified by the system parameter maximum client salutation size. If multiple salutations are employed, the service schedule for a particular salutation may be changed using CSS.

The message forwarding function enables the client to direct the system **100** to place a telephone call to him when messages are received in his message basket. This call is handled in accord with its delivery instructions and the client service options in effect.

If immediate forwarding is specified, the system **100** attempts to forward the message within the number of minutes specified by the immediate forwarding interval. Otherwise, forwarding of messages that the client has not heard shall only be attempted during forwarding period established through customer service. The system **100** accepts forwarding instructions either directly from the client or from the client via an information operator. The system **100** establishes a forwarding condition for a client upon entry by the client of a forwarding on command. Upon receiving this command, the system **100** responds with a prompt giving either:

1. The previously established forwarding telephone number or message basket number and request confirmation of the number or
2. Requesting the forwarding number.

The client either confirms the existing number by entering the save command or enter the number using a data syntax as described for edit/accept delivery instructions. If the client enters a number, the system **100** will respond with a prompt that repeats the number for his confirmation. The system **100** deactivates a forwarding condition for a client upon the client's entry of a forwarding command. The daily start/stop time, if any, and the number of attempts and retry interval associated with the client controlled forwarding is established through CSS. CSS provides a client the capability to specify a maximum of five sets of message forwarding instructions to the system **100**. An instruction contains the forwarding phone number, start date and time, end date and time of retry interval and number of retries.

The message delivery function shown in FIG. **14** delivers a message to a message basket or telephone number. As illustrated in greater detail in FIG. **26**, delivery to a telephone number may be done either automatically or by operator assistance. Delivery is accomplished with operator assistance unless automatic delivery is intentionally selected by the client. If verification or identification of the called party is required, then operator assisted delivery is required.

For deliveries requiring operator assistance, the system brings an operator on-line and automatically dials the telephone number selected by the operator. When the call is answered, the system **100** information operator verifies the identity of the called party, initiates the message playout, and executes any special instructions which have been directed by the client.

For automatic delivery, system **100** dials the addressee. At the time system **100** determines that the call has been answered, the delivery prompt is played to the called party. This prompt states that there is a recorded message for delivery and requests an acknowledgement from the called party before beginning playout of the

message. The message is played to the answering party upon receipt of this acknowledgement from the answering party. If this command is not received within a prompt interval, the system **100** directs connection to an information operator for assistance. The system **100** provide a client option allowing automatic delivery without acknowledgement.

In cases where a called telephone number is not answered on the first attempt, the system **100** continues to attempt delivery at intervals and for the number of retries established in delivery instructions. In addition, most delivery attempts resulting in a "busy" are retried in the number of minutes specified in the busy retry interval to the maximum specified by busy retry count. The client has the option to allow the called party to respond to a message at the time of delivery. In this case, the system **100** plays the reply invitation prompt. The called party is able to record a single message. The reply message is left in the client's message basket. The capability for called party response is selectable for each addressee in the subscriber's delivery instructions for either operator assisted or automatic telephone delivery.

Messages which are undeliverable are flagged within the sending subscriber's message basket. These messages and their status are played out by the retrieve message function. For automatic delivery that does not require or does not result in information operator assistance, a message a considered as delivered when the system **100** has determined that the call has been answered or acknowledgement received.

INTERFACE ROUTINES

The following interface routine functional specification is useful in helping a skilled programmer to understand and utilize the program listings set forth in Table I.

Process management in the virtual machine consists of a set of interface routines to REX process management. The use of function codes and their associated jump tables have been defined to fit in a Pascal environment and some of the options available to REX callers have been combined or eliminated for this release at the Pascal level process management.

The process management procedures and functions which the virtual machine provides include:

(1) VCREATE—create a new subprocess.

(2) VCRNODE—create a node of a job.

(3) VQUIT—terminate the calling process.

(4) VCREATOR_PID—return the PID of the creator of the calling process.

(5) VDECLARE_FC—declare the valid packet function codes for the calling process.

(6) VSELF_PID—return the PID of the calling process.

VCREATE—create a new subprocess is system privileged and has the following procedure definition: function VCREATE (var SPID: vspidrange; PRTYCLASS: vclassrange; PRTYRANK: vrankrange; PGMID: vprogid; PGMNONSHARE: boolean): vbit 16.

Among the inputs to VCREATE, SPID is the specific subprocess ID to create or zero if the system should assign one. PRTYCLASS is the priority class, from 1 to 4, that will be assigned to the created process. PRTYRANK is the priority rank within the class, from 1 to 255, that will be assigned to the created process. PGMID is the system program ID of the Pascal program to run in the created process. PGMNONSHARE is a boolean flag indicating whether the code specified in PGMID is shareable or not. A true value indicates nonshareable code. Among the outputs, SPID contains the subprocess ID of the new subprocess.

VCREATE returns as VCOMPOK if the create was successful; VMEMUNAVAIL if sufficient extended memory to create the process was not available; VSPIDUNAVAIL if a REX subprocess was not available; or a REX event completion code if a REX system error occurred.

If a nonzero SPID is specified, that SPID must be available; if zero is specified, some SPID must be available.

It should be noted that:

(1) This function calls the REX routine CREATE#.

(2) All virtual machine subprocesses require the assignment of a data segment. There are 255 such segments in any one GPP, some of which are used by the virtual machine and some are used for code. This leaves a practical limit of 250 processes running programs in a GPP.

(3) The named program will be loaded if it does not already exist in memory and its user count will be incremented if it already has been loaded.

(4) Execution of the process starts at the main entry point of the program.

VCRNODE creates a node of a job by sending a create node request packet to JSAM and access is system privileged. The procedure definition is procedure VCRNODE (var PKT: vpacket; var CRNRESULT: vcomplcode; var RPKT: vpacket). PKT is the create node request packet to be sent to JSAM. Ihe header will be filled in by this procedure which will also wait for a response. CRNRESULT is the event completion code from the create node. A value other than VCOMPOK is considered an error. RPKT is the response packet from JSAM for the create node request. All requirements applicable to creation of a node of a job, as stated in the JSAM documentation, apply to this procedure.

VQUIT terminates the calling process and is system privileged. The procedure definition is procedure VQUIT (PDSCLEANUP:boolean). PDSCLEANUP is true if the procedure has been called by PDS to cleanup resources between commands. There are no outputs. This function has requirements similar to the QUIT function. This procedure is intended to be called by SYS.PDS and SYS.DRVR to end the execution of a program. In the case of SYS.DRVR, PDSCLEANUP is false which calls for a full process quit. In the case of SYS.PDS, when the user segment completes execution, PDSCLEANUP is set true to cause the resources for the user program to be released so the next user program can be run. This procedure calls the REX routine QUIT.

VCREATORPID returns the PID of the creator of the calling process and is system privileged. Procedure definition is VCREATORPID (var PID: vpidrec). There are no inputs and PID identifies the creator of the calling process. This procedure calls the REX routine CREATORIPID.

VDECLAREFC declares the valid packet function codes for the calling process and is system privileged. The procedure definition is VDECLAREFC (FCMASK: vfcset). FCMASK is a set with the valid function codes from 0 to 15 set. There are no outputs. Note that DECLAREFC calls the REX routines SETFCMASK and RESTFCMASK. This procedure

declares the valid packet function codes that will be returned to the Pascal program. The program then has to handle the functions itself through a case statement or similar construct.

VSELFPID returns the PID of the calling process and is system privileged. The procedure definition is VSELFPID (var PID: vpidrec). It outputs the PID of the calling process returned in PID.

Event management in the virtual machine consists of a set of interface routines to the higher level REX event management routines. Event management deals solely with the detection of events and not the allocation or deallocation of event control blocks (ECBs). To the user of the virtual machine, ECBs are an internal structure used totally by the system to maintain events for the user. The event management procedures and functions which the virtual machine provides include:

(1) VCHECK_EVENT—check for the occurrence of an event and return true if the event has occurred, otherwise return false.

(2) VWAIT—make the calling process nondispatchable until an associate event occurs.

VCHECK_EVENT checks for the occurrence of an event and return true if the event has occurred; otherwise, return false. It is unprivileged. The procedure definition is VCHECK_EVENT (EVENTIDTYPE: vchecktype; var USERREFID: vuserrefval; RESPONSEID: vecbid; var EVENTCODE: vcomplcode; var CHECKPKT: vpacket): boolean. As inputs EVENTIDTYPE specifies whether any event or a specific event by user reference value or response ID is being checked. The following are valid values for this parameter:

(1) VCHKANY—check for any event for this process.

(2) VCHKUSERREF—check for a specific event with a user reference value specified in USERREFID.

(3) VCHKRESPONSEID—check for a specific event with a response ID value specified in RESPONSEID.

USERREFID is the user reference value of a specific user event to be checked. RESPONSEID is the ECBID of a specific event to be checked. If USERREFID is used, then RESPONSEID is not necessary. As outputs USERREFID is returned with the user reference value (this does not apply to pure timers). EVENTCODE is the event completion code as defined in the appendices.

CHECKPKT is a 16-word packet that is returned if the associated event returns a packet. This function calls the REX routines CHECKEDB#3, CHECKEDB$E#3, or CHECKECB$U#3 depending on the options that are selected. The user reference value returned is that which was specified when the event was defined. Packets returned to unprivileged processes will have the PID fields of the header zeroed.

VWAIT makes the calling process nondispatchable until an associate event occurs and is unprivileged. The procedure defintion is VWAIT (WAITTYPE: vwaittype; var USERREFID: vuserrefval; RESPONSEID: vecbid; var EVENTCODE: vcomplcode; var WAITPKT: vpacket). For inputs WAITTYPE specifies whether a VWAITANY, for waiting on any event, or a VWAITSPECIFIC, to wait on a specific event, is required. RESPONSEID is the ID associated with a specific user event that is being waited on. This is ignored for VWAITANY. For outputs WAITPKT is a 16-word packet that is returned if the associate event returns a packet, EVENTCODE is the event completion code as defined in the appendices, and USERREFID is the ID of a specific user event that is being returned. This procedure calls the REX routines WAIT or WAIT$E depending on the options that were selected. If the specified event (or any) has already occurred, the process remains dispatchable. Control is not returned, however, until the associated PCB again comes to the head of the dispatchable queue.

Time management in the virtual machine is a set of procedures offered to interface with the system time management. For details on the system time management, refer to the REX subsystem specification.

The time management procedures and functions which the virtual machine provides include:

(1) VSTART_TIMER—Start a REX timer for the current running process.

(2) VCANCEL_TIMER—Cancel a timer that was set for this process.

(3) VDATETIME—Return the current date and time to the caller.

VSTART_TIMER starts a REX timer for the current running process and is unprivileged. The procedure definition is VSTART_TIMER (var RESPONSEID: vecbid; UREFID: vuserrefid; TIMELTH: integer; TIMEUNIT: vmmtimeunits; TIMEWAIT: boolean). For inputs UREFID is the user reference value that is established by the user and may be used to identify the expiration of the timer. If TIMEWAIT is true, this field is not used. TIMELTH is the timer interval expressed in milliseconds or seconds. TIMEUNIT specifies the interval type as mSEC or SEC. TIMEWAIT is a boolean requesting suspension of the process until the timer expires if true and immediate return if false. For outputs RESPONSEID is a REX event ID that is returned if wait was not requested. This ID is used when calling CHECKEVENT or WAIT. This procedure calls the REX routines STARTTIMER or STARTTIMER$S, depending on the value of TIMEUNIT. If the timer interval is expressed in milliseconds, the largest interval is 65,535 milliseconds or 1 minute, 5 seconds, 535 ms. When expressed in seconds, the largest interval is 18 hours, 12 minutes, 15 seconds. A timer may only be started for the active process.

VCANCEL_TIMER cancels a timer that was set for this process and is unprivileged. The procedure definition is VCANCEL_TIMER (RESPONSEID: vecbid). RESPONSEID is input as the REX event ID associated with the timer being cancelled. This procedure calls the REX routine CANCELTIMER. A timer may be cancelled only if it is associated with the active process.

VDATETIME returns the current date/time to the caller and is unprivileged. The procedure definition is VDATETIME (CDATETIME: vdtrectype): boolean. The function VDATETIME returns true as an output if the current date is set in the processor; otherwise, it returns false. CDATETIME is set to the current date/time as a packed record of the following definition if the current date is set in the processor.

```
vdtrectype=packed record
  YEAR: 0..99;
  MONTH: 1..12;
  DAY: 1..31; (*Day of Month*)
  DOY: 1..366; (*Day of Year*)
  DOW: 1..7; (*Day of Week*)
  HOUR: 0..23;
  MIN: 0..59;
```

SEC: 0..59;
HSEC: 0..99; (*Hundreths of Seconds*)

This routine refers to the date maintained by REX in page zero.

Job management in the virtual machine handles all communication with JSAM for creation and deletion of processes that exist as nodes of a job in the JSAM sense. The levels of interface between the virtual machine subsystem and JSAM include:

1. Processing in GPEXREX for handling incoming packets from JSAM to control main processes that are implemented as virtual machines; and

2. Processing in the VMM for handling requests from JSAM to control subprocesses that run as virtual machines in the GPP.

All processes in the GPP that are implemented as virtual machines are managed by REX/GPEXREX and are identified as virtual machines to allow the special handling necessary to control them. Internally, in the GPP, and externally they are referred to as "special processes" to distinguish them from processes that are controlled strictly by REX. All requests to create special processes are rounted to GPEXREX which decides whether the request is for a system process to be run as a main process or a nonsystem process to be run as a subprocess of a VM controlling process. For the first release, nonsystem processes will include TASS and PDS application whereas system processes will include SYSDLO, SYSDEV, and SYSMON.

The processing involved in processing these packets includes:

1. Create process request packet. Sent by JSAM to request the creation of a virtual machine process.

2. Create process response packet. Sent by VMM to JSAM to report on the status of a create request.

3. Delete process request packet. Sent by JSAM to request the deletion of a virtual machine process.

4. Delete process report packet. Sent by VMM to JSAM to report the completion normally or abnormally of a virtual machine process.

I/O control in the virtual machine is performed by a number of input/output services routines (IOSRs), each of which interfaces with a REX IOSR to perform a prescribed I/O operation. The IOSRs operate as an extension of the user program and establish control and data paths between the user and the rest of the system **100**.

In general, an I/O operation proceeds as follows. A user requests that an I/O function be performed by invoking the appropriate IOSR with a set of parameters that define the request. The IOSR formats the parameters into REX IOSR calling arguments and calls the appropriate REX IOSR. At that point the user's process is suspended until the operation is complete. Upon return from the request, VM IOSRs either return with a successful completion or return the appropriate status to indicate why the operation was unsuccessful.

In this sequence, all I/O is performed on behalf of the user's process and the associated environment. Suspension of the process takes place if sufficient resources are not currently available for REX to initiate the operation. Then, the process waits for completion of the I/O operation before being redispatched for return to VM IOSRs and the user. The following paragraphs describe the VM IOSR procedures in general as they apply to all device and data set handlers.

The VM IOSRs support the acquiring and releasing of devices through various procedures. When a device is acquired, an acquire response packet is returned which contains the information that is required to open and access the device. To accommodate this, the VM IOSR maintains these packets in its environment with an associated unique identifier. This identifier is then placed in the user's file information block (FIB) for future reference when required by VM IOSRs, but the actual packet is not made available to the user at any time.

All devices that are acquired and all data sets that are open must have a unique FIB associated with them. The FIB resides in the user's data space and contains descriptive information about the device or data set. This information is used by the VM IOSRs to interface with system functions when executing I/O procedures for the user and must exist from acquire to release for devices and open to close for data sets.

It is defined as follows:

```
Vfib=packed record
  VCHANREF1: vbit 8;
  VCHANREF2: vbit 8;
  VDEVNAME1: vdevname;
  VDEVNAME2: vdevname;
  VSYSCOMPL: vcomplcode;
  VDEVCOMPL: vdevcompcode;
  VACQPKTID: vbit 16;
  VTIMEOUT: vbit 8;
  VPRIORITY: vbit 8;
end (Vfib);
```

where:

1. VCHANREF1 is the reference number of the primary channel on a multichannel device or the reference number of the only channel on a single channel device.

2. VCHANREF2 is the channel number of the second channel on a multichannel device.

3. VDEVNAME1 is the device name associated with VCHANREF1 for the device being accessed.

4. VDEVNAME2 is the device name associated with VCHANREF2 for the device being accessed.

5. VSYSCOMPL is the REX event completion code.

6. VDEVCOMPL is the device completion code which is made up of the device error code and the device error group.

7. VACQPKTID is the internal ID of the acquire packet which is maintained by VM IOSRs. This ID is established when a device is acquired and must remain intact until the device is released.

8. VTIMEOUT is the time (in seconds) to be added to the system timeout value to arrive at the timeout interval for the operation.

9. VPRIORITY is the priority to be assigned to the associated operation and is required only if it is implemented by the target handler.

The access control block (ACB) is used to maintain information relevant to a particular access of a device or data set. As such, at any given time, a unique ACB must exist for each active access. An access is established by the execution of one of the access-type verbs (e.g., VREAD, VWRITE . . .) and remains active until the access is completed (usually by a VENDIO operation).

The ACB is defined as follows:

```
vacb=packed record
  VACCESSREF: vbit 16;
  VACBCOMPL: vcomplcode;
  VACBRESP: vioreturnpacket;
  VTIMEOUT: vbits 8;
  VPRIORITY: vbit 8
```

end (vacb);
where:

1. VACCESSREF is a unique reference ID assigned and used by VM IOSRs to associate the various functions executed as part of the specific access. Modification of this field by the user will cause the access to be aborted.

2. VACBCOMPL is the REX event completion code related to the last operation performed on this access.

3. VACBRESP is a 10-word area where the last 10 words of the response packet, related to the last operation for this access is returned.

4. VTIMEOUT is the time (in seconds) to be added to the system timeout value to arrive at the timeout interval for the operation.

5. VPRIORITY is the priority to be assigned to the associated operation and is required only if it is implemented by the target handler.

VACQNAME acquires ownership of a system device by its iternal name and is unprivileged. The procedure definition is VACQNAME (var VMFIB: vfib; DEVNAME: vdevicename; VMIOFLAGS: vioflags; USNFUNCCODE: vfcrange). For inputs:

1. VMFIB will have the following fields set:
   a. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.
   b. VPRIORITY is the priority value to be assigned to the acquire operation.
2. DEVNAME is the internal device name of the device being acquired.
3. VMIOFLAGS is a set of flags that specify the selected acquire options as follows:
   a. VACQQUEUE is set if SYSDEV is permitted to queue the acquire.
   b. VACQSEIZE is set if one of the device channels is to be seized during the acquire processing.
4. USNFUNCCODE is the function code to be used by the system for unsolicited notification packets. For outputs:

VMFIB will have its fields set as follows:

VCHANCNT will contain the number of channels that the acquired device has.

VCHANINFO[I].VCHANREF contains channel reference values (1–255) that will be used by the VM IOSRs in all subsequent I/O operations for the associated channel. Note that I ranges from 0 to VCHANCNT.

VCHANINTO[I].VFDEVNAM contains the internal device names of the device for the associated channel and will be used by the VM IOSRs in all subsequent I/O operations.

VSYSCMPL contains the REX event completion code.

VDEVCMPL contains the acquire completion code.

VACQPKTID contains the ID of the acquire response packet received from SYSDEV if the device has been successfully acquired. The ID will be used by the VM IOSRs to access the acquire response packet when an I/O operation requires information contained in the packet.

VDEVSTATUS will be set to VAQUIRED if the operation was successful.

Upon calling VACQNAME, the user process is suspended until successful or unsuccessful completion of the acquire function. VSYSCOMPL and VDEVCOMPL fields in the FIB indicate the result of the acquire function. The user process must not alter any of the values returned in the FIB. This procedure sends an acquire request packet to SYSDEV.

VACQLIST acquires ownerhip of a system device by type and subtype and is unprivileged. The procedure definition is VACQLIST (var VMFIB: vfib; DTYPE: vdevtype; SUBTYPE: vstypelist; VMIOFLAGS: vioflags; USNFUNCCODE: vfcrange). For inputs:

1. VMFIB will have the following fields set:
   a. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.
   b. VPRIORITY is the priority value to be assigned to the acquire operation.
2. DTYPE is the system device type of the device being acquired.
3. SUBTYPE is a list of acceptable system subtypes for the specific device type being acquired.
4. VMIOFLAGS is a set of flags specifying the selected acquire options as follows:
   a. VACQQUEUE is set if SYSDEV is permitted to queue the acquire.
   b. VACQSEIZE is set if one of the device channels is to be seized during the acquire processing.
5. USNFUNCCODE is the function coce to be used by the system for unsolicited notification packets.

For outputs VMFIB has its field set as follows:

1. VCHANINFO[I].VCHANREF contains channel reference values (1–255) which the VM IOSRs use in all subsequent I/O operations for the associated channel. Note that I ranges from 0 to VCHANCNT.
2. VCHANINTO[I].VFDEVNAM contains the internal device names of the device, and the VM IOSRs use it in all subsequent I/O operations on channel one.
3. VSYSCOMPL contains the REX event completion code.
4. VDEVCMPL contains the acquire completion code.
5. VACQPKTID contains the ID of the acquire response packet received from SYSDEV if the device has been successfully acquired. The VM IOSRs use the ID to access the acquire response packet when an I/O operation requires information contained in the packet.
6. VDEVSTATUS will be set to VACQUIRED.

Upon calling VACQLIST, the user process is suspended until successful or unsuccessful completion of the acquire function. VSYSCOMPL and VDEVCOMPL fields in the FIB indicate the result of the acquire function. The user process must not alter any of the values returned in the FIB. This procedure sends an acquire request packet to SYSDEV.

VOPEN opens a logical path to a device and is unprivileged. The procedure definition is VOPEN (var VMFIB: vfib; VMIOFLAGS: vioflags).

For inputs VMFIB will have the following fields set:

1. VCHANINFO as set by the VM IOSR acquire ownership operation.
2. VACQPKTID as set by the VM IOSR acquire ownership operation.
3. TIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

VMIOFLAGS will have the following set: VCHANNO is the number of the channel to be opened for multiple channel devices. It should be zero for devices with only one channel.

For outputs VMFIB will have the following fields returned:

1. VSYSCOMPL is the REX event comp1etion code.

2. VDEVCOMPL is the open device completion code.

3. VDEVSTATUS will be set to VOPENED if the operation was successful.

The FIB must contain the values returned from the acquire of the device being opened. This procedure calls the REX routine OPEN#.

VOPEN DSET opens a logical path to a data set and is unprivileged. The procedure defintion is VOPENDSET (var VMFIB: vfib; DSNAMLEN: vbit 8; DSNAMPTR: vaddr; var DSINFO: vdatasetinfo).

For inputs:

1. VMFIB will have the following fields set:

a. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

b. VPRIORITY is the priority value to be assigned to the operation.

3. DSNAMLEN is the length (in bytes) of the data set name.

4. DSINFO is a packed record whose fields specify access restrictions and default operational characteristics.

5. DSINFO will have the following fields set:

a. VDSEXCL is set if this open is for exclusive access. If the flag is rest, this open is for shared access.

b. VDSRDWR is set if READ/WRITE is allowed for the data set; otherwise, (flag is reset) open is for read only.

c. VDSUTIL is set for utility open and is reset for open without utility privileges.

d. VDSUNCAT is set if catalog is not to be used to locate data set. If the flag is reset, catalog is used to locate data set.

e. VDSUP is set if data set is duplicated; otherwise, (flag is reset) data set is singular.

f. VUSER is packed array of 5 bytes, specifying user code of requestor. (This field is currently unused.)

g. VOLID1, VOLID2 are packed arrays of six chars, specifying volume ID of the volume on which data set resides.

For outputs

VMFIB will have its fields set as follows:

1. VCHAINFO[0].VCHANRF contains a channel reference value (1–255) that the VM IOSRs use in all subsequent I/O operations on the opened data set.

2. VSYSCOMPL contains the REX event completion code.

3. VOPENRESP contains the open response packet itself.

If the catalog is to be used to locate the data set (VDSUNCAT=1) the volume ID of the volume on which the data set resides is required (VOLID1). Otherwise, this field can be omitted. (If the data set is duplicated [VDSUP=1], this field must specify the volume ID of the volume on which the primary copy of the data set resides.) If the data set is duplicated (VDSUP=1) and the catalog is not to be used to locate the data set (VDSUNCAT=1), this field must specify the volume ID of the volume on which the secondary copy of the data set resides (VOLID2). Otherwise, this field can be omitted.

VREADCRT builds an access path to a logical data record in a CRT to permit transfer of data from the CRT handler to the user and is unprivileged. The procedure definition is VREADCRT (var VMFIB: vfib; var VMACB: vacb; CRTCNTL: vcrtdctrl); VMIOFLAGS: vioflags).

For inputs VMFIB will have the following field set:

1. VCHANINFO as set by the VM IOSR acquire ownership operation.

CRTCNTL is a record that will have the following fields set:

1. VSOM is a start-of-message character that specifies the type of read.

2. VPARTNO is a character that specifies the CRT partition number from which data will be read.

3. VLINEITEM is a character that specifies the line number with the partition number from which data will be read.

4. VEOM is a character that specifies the end-of-message character.

VMIOFLAGS will have the following field set:

1. VCHANNO is the number of the channel to be accessed for multiple channel devices. It should be zero for single channel devices.

For outputs VMACB will have its fields set as follows:

1. VACCESSREF contains an access reference value (1–255) that the VM IOSRs use in all subsequent I/O operations related to this access path.

2. VACBCOMPL contains the REX event completion code.

3. VACBRESP is the response returned from the device handler that will have its fields set as follows:

a. VDEVCC contains the status returned from the device handler.

b. VDATALEN contains the size (in bytes) of the record being accessed.

The FIB must contain the values returned by the procedure OPEN for the device being read. Multiple accesses to a single CRT are supported. If the handler cannot support this, it is the handler's responsibility to reject the request. The user process must not alter any of the values returned in the ACB. This procedure calls the REX routine ACCESS#.

VREAD DIRECT builds an access path to a logical record in a direct data set to permit transfer of data between the user and the data set handler and is unprivileged. The procedure definition is VREADDIRECT (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags; ELEMNO: vlonginteger).

For inputs:

VMFIB will have the following field set:

1. CHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following field sets:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VREADDIRECT options as follows:

1. VUPDATE is set if this is an updating read. The record element will be locked against other accesses until released with an End I/O request.

ELEMNO is the record element of the record to be accessed.

For outputs:

VMACB will have its field set as follows:

1. VACCESSREF contains an access reference value (1–255) that the VM IOSRs use in all subsequent I/O 0 operations related to this access path.

2. VACBCOMPL contains the REX event completion code.

3. VACBRESP is the response packet returned by the data set handler that will have its fields set as follows:

a. VDEVCC contains the error/code group.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1–16) of the packet.

The FIB must contain the values returned from the procedure OPENDSET for the data set being read. The user process must not alter any of the values returned in the ACB. This procedure calls the REX routine ACCESS#.

VREADKEY builds an access path to a logical record in an indexed data set to permit transfer of data between the user and the data set handler and is unprivileged. The procedure definition is VREADKEY (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags; KEYLENGTH: vbit8; KEYPTR: vaddr).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operations.

2. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VREADKEY options as follows:

1. VUPDATE is set if this is an updating read. The record will be locked against other accesses until released with an End I/O request.

2. VGREATER is applicable when the flag VKAPPROX is set. If VGREATER is set, the retrieval key will be the first key strictly greater than the user key. If VGREATER is not set, the retrieval will be that record whose key is the first key in the index lexically greater than or equal to the key supplied by the user.

3. VDATAREC is set if the data record is not to be retrieved. If the flag is rest, the data record is retrieved.

4. VKAPPROX is set if the read is an approximate read. (See VGREATER flag definition).

KEYLEN is the length (in bytes) of the record key.

KEYPTR is a pointer to a packed array of characters that contains the name of the record key.

For outputs:

VMACB will have its fields set as follows:

1. VACCESSREF contains an access reference value (1–255) the VM IOSRs use in all subsequent I/O operations related to this access path.

2. VACBCOMPL contains the REX event completion code.

3. VACBRESP is the response packet returned by the data set handler that will have its fields set as follows:

a. VDEVCC contains the error code/error group.

b. Other fields in this packet are unnamed but may be accessed by VPKINIT[N], where N is the word number (1–16) of the packet.

The FIB must contain the values returned by the procedure VOPENDSET. The user process must not alter any of the values returned in the ACB. This procedure calls the REX routine ACCESS#.

VREADNEXT builds an access path to a logical record in an indexed data set. The record will be retrieved whose key is lexically next greater than the most recent previously completed access on this channel and is unprivileged. The procedure definition is VREADNEXT (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the systems timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VREADNEXT options as follows:

1. VUPDATE is set if this is an updating read. The record will be locked against other accesses until released with an End I/O request.

2. VNODATA is set if the data record is not to be retrieved. If the flag is reset, the data record is retrieved.

3. VSAMREC is set if the access is to remain on the same record. If the flag is reset, access the next record.

4. VNULL is set for null continue; otherwise, (flag is reset) normal continue.

For outputs:

VMACB will have its fields set as follows:

1. VACCESSREF contains an access reference value (1–255) that the VM IOSRs use in all subsequent I/O operations related to this access path.

2. VACBRESP is the response returned by the data set handler that will have its fields set as follows:

a. VDEVCC contains the error code/error group.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1–16) of the packet.

The FIB must contain the values returned by the procedure VOPENDSET. The user process must not alter any of the values returned in the ACB. This procedure calls the REX routine ACCESS#.

VCONTINUE releases the current record and prepare to read, write or update the next record in a block data set and is unprivileged. The procedure definition is VCONTINUE (var VMFIB: vfib; var VMACB: vacb).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the operation.

For outputs:

VMACB will have its fields set as follows:

1. VACBCOMPL contains the REX event completion code.

2. VACBRESP is the response packet returned by the data set handler that will have its fields set as follows:

a. VDEVCC contains the status returned from the device or data set handler.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1–16) of the packet.

The FIB must contain the values returned by the procedure VOPENDSET. The user process must not alter any of the values returned by the procedure VOPENDSET. This procedure calls the REX routine ACCESS#.

VWRITEDEV builds a write access path to a logical data record to permit transfer of data from a user to a device and is unprivileged. The procedure definition is VWRITEDEV (var VMFIB: vfib; var VMACB: vacb; NBYTES: integer; VMIOFLAGS: vioflag).

For inputs:

VMFIB will have the following fields set:

1. VCHANINFO as set by the VM IOSR acquire ownership operation.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the read operation.

NBYTES is the number of bytes to be set up for this write device access.

VIOFLAGS will have the following field set:

1. VCHANNO is the number of the channel to be accessed for multiple channel devices. It should be zero for single channel devices.

For outputs:

VMACB will have the following fields set on return:

1. VACCESSREF is a unique value assigned by VM IOSRs to be used throughout the life of the write access sequence.

2. VACBCOMPL is the event completion code returned by REX.

3. VACBRESP is the response returned by the device handler that will have its fields set as follows:

   a. VDEVCC contains the status returned from the device handler.

The FIB must contain the values returned by the procedure OPEN for the device being written to. This procedure calls the REX routine ACCESS#. Before any data can be transferred from the user to the device handler, both parties must be made ready for the transfer by a VWRITEDEV operation. If the device being written to is a Zentec, only one access may be active at a time.

VWRITEDIRECT builds an access path to a logical record in a direct data set to permit transfer of data from the user to the data set handler and is unprivileged. The procedure definition is VWRITEDIRECT (var VMFIB: vfib; var VMACB: vacb; ELEMNO: vlonginteger).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the operation.

ELEMNO is the record element number (also relative to the start of the dataset) of the record to be written. This parameter applies to disk data sets only.

For outputs:

VACCESSREF contains an access reference value (1–255) the VM IOSRS use in all subsequent I/O operations related to this access path.

VACBCOMPL contains the REX event completion code.

VACBRESP is the response returned by the data set handler that will have its fields set as follows:

1. VDEVCC contains the error code/group code.

2. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1–16) of the packet.

The FIB must contain the values returned by the procedure VOPENDSET. The user process must not alter any of the values returned in the ACB. This procedure calls the REX routine ACCESS#.

VWRITEKEY builds access path to a logical record in an indexed data set to permit transfer of data from the used to the data set handler and is unprivileged. The procedure definition is VWRITEKEY (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags; KENLENGTH: vbit8; KEYPTR: vaddr).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VWRITEKEY options as follows:

1. VREPLACE is set if this Write is meant to replace an existing record, and an error is implied if the record does not exist. If the flag is reset, the Write is meant to create a new record, and the presence of an existing record with the same key implies an error.

KEYLEN is the length (in bytes) of the record key.

KEYPTR is a pointer to the record key.

For outputs:

VMACB will have its fields set as follows:

1. VACCESSREF contains an access reference value (1–255) the VM IOSRs use in all subsequent I/O operations related to this access path.

2. VACBCOMPL contains the REX event completion code.

3. VACBRESP is the response packet returned by the data set handler that will have its fields set as follows:

   a. VDEVCC contains the error code/group code.

   b. Other fields in this packet are unnamed, but may be accessed VPKTINIT[N], where N is the word number (1–16) of the packet.

The FIB must contain the values returned by the procedure VOPENDSET. The user process must not alter any of the values returned in the ACB. This procedure calls the REX routine ACCESS#.

VGET transfers data from a device or data set handler buffer to a user buffer and is unprivileged. The procedure defintion is VGET (var VMFIB: vfib; var VMACB: vacb; BUFFPTR: vaddr; SOURCEOFFSET: integer; DESTOFFSET: integer; NBYTES: integer).

For inputs:

VMFIB will have the following fields set:

1. VCHANINFO[O].VCHANRF as set by the or Open data set operation.

VMACB will have the following fields set:

1. VACCESSREF is the value assigned by VM IOSRs to uniquely identify the access path that was set up by the previous Read.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for this operation.

3. VPRIORITY is the priority value to be assigned to the get operation.

BUFFPTR is the address of the buffer, in the user's space, where the get will return data.

SOURCEOFFSET is the byte offset into the record from which point the data is to be read.

DESTOFFSET is the byte offset into the user's buffer where the data is to be transferred.

NBYTES is the number of bytes to be transferred.

For outputs:

VMACB will have the following fields set on return:

1. VACBCOMPL is the event completion code returned by REX.

2. VACBRESP is the response returned by the device or data set handler that will have its fields set as follows:

a. VDEVCC contains the status returned from the device or data set handler.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1-16) of the packet.

The VMFIB must contain the values returned by the procedure VOPEN or VOPENDSET. The access path specified by the VACCESSREF must exist, i.e., the user must have previously called one of the VREAD procedures. The only range checking that will be performed on the get operation is to ensure that the entire buffer will fit in the user's address space. Therefore, a specification to get more bytes of data into a variable than the size of the variable will result in overwriting an area outside the variable. This will most often cause the program to malfunction. This procedure calls the REX routine GET#.

VWGET transfers data from a DSR buffer to a user's local buffer. (This function is applicable for indexed data sets.) The function is unprivileged. The procedure defintion is VWGET (var VMFIB: vfib; var VMACB: vacb; BUFFPTR: vaddr; SOURCEOFFSET: integer; DESTOFFSET: integer; NBYTES: integer; VMWHERE: vbit2).

For inputs:

The VMWHERE field can be set to the following:

1. VDATAFLD (=0): get user data record;

2. VMEMOFLD (=1): get memo record;

3. VKEYFLD (=2): get key.

VMFIB will have the following fields set:

1. VCHANINFO[O].VCHANRF as set by the VM IOSRs to uniquely identify the access path that was set up by the previous Read.

VMACB will have the following fields set:

1. VACCESSREF is the value assigned by VM ISORs to uniquely identify the access path that was set up by the previous Read.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for this operation.

3. VPRIORITY is the priority value to be assigned to the get operation.

BUFFPTR is the address of the buffer, in the user's space, where the get will return data.

SOURCEOFFSET is the byte offset into the record from which point the data is to be read.

DESTOFFSET is the byte offset into the user's buffer where the data is to be transferred.

NBYTES is the number of bytes to be transferred.

For outputs:

VMACB will have the following fields set on return:

1. VACBCOMPL is the event completion code returned by REX.

2. VACBRESP is the response packet returned by the source device or data set handler that will have its fields set as follows:

a. VDEVCC contains the status returned from the device or data set handler.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1-16) of this packet.

The VMFIB must contain the values returned by the procedure VOPEN or VOPENDSET. The access path specified by the VACCESSREF must exist, i.e., the user must have previously called one of the VREAD procedures. The only range checking that will be performed on the get operation is to ensure that the entire buffer will fit in the user's address space. Therefore, a specification to get more bytes of data into a variable than the size of the variable will result in overwriting an area outside the variable. This will most often cause the program to malfunction. This procedure will call the REX routine GET#.

VPUT transfers data from a user's buffer to a device or data set handler buffer and is unprivileged. The procedure definition is VPUT (var VMFIB: vfib; var VMACB: vacb; BUFFPTR: vaddr; SOURCEOFFSET: integer; DESTOFFSET: integer; NBYTES: integer).

For inputs:

VMFIB will have the following fields set:

1. VCHANINFO[O].VCHANRF as set by the Open device or Open data set operation.

VMACB will have the following fields set:

1. VACCESSREF is the value assigned by VM ISORs to uniquely identify the access path that was set up by the previous write.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for this oeration.

3. VPRIORITY is the priority value to be assigned to the put operation.

BUFFPTR is the address of the buffer, in the user's space, where the put will get its data.

SOURCEOFFSET is the byte offset, into the user's buffer, where the effected data starts.

DESTOFFSET is the byte offset, into the handler's record or record element where the data is to be transferred.

NBYTES is the number of bytes to be transferred.

For outputs:

VMACB will have the following fields set on return:

1. VACBCOMPL is the event completion code returned by REX.

2. VACBRESP is the response returned by the device or data set handler that will have its field set as follows:

a. VDEVCC contains the status returned from the device or data set handler.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1-16) of the packet.

The FIB must contain the values returned by the procedure VOPEN on VOPENDSET. The access path

specified by the VACCESSREF must exist, i.e., the user must have previously called on the write procedure. This procedure calls the REX routine PUT#.

VWPUT transfers data from a user's buffer to a DSR buffer. (This function is applicable for indexed data sets.) This function is unprivileged. The procedure definition is VWGET (var SMFIB: vfib; var VMACB: vacb; BUFFPTR: vaddr; SOURCEOFFSET: integer; DESTPFFSET: integer; NBYTES: integer; VMWHERE: vbit2).

For inputs:

The VMWHERE field can be set to the following:

1. VDATAFLD (=0): put user data record;
2. VMEMOFLD (=1): put memo record;
3. VKEYFLD (=2): put key.

VMFIB will have the following field sets:

1. VCHANINFO[O].VCHANRF as set by the VM ISORs to uniquely identify the access path that was set up by the previous Write.

VMACB will have the following fields set:

1. VACCESSREF is the value assigned by VM ISORs to uniquely identify the access path that was set up by the previous Write.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for this operation.

3. VPRIORITY is the priority value to be assigned to the put operation.

BUFFPTR is the address of the buffer, in the user's space, where the put will get its data.

SOURCEOFFSET is the byte offset, into the user's buffer, where the effected data starts.

DESTOFFSET is the byte offset into the handler's record or record element, where the data is to be transferred.

NBYTES is the number of bytes to be transferred.

For outputs:

VMACB will have the following fields set on return:

1. VACBCOMPL is the event completion code returned by REX.

2. VACBRESP is the response packet returned by the source device or data set handler that will have its fields set as follows:

a. FDEVCC contains the status returned from the device or data set handler.

b. Other fields in this packet are unnamed, but may be accessed by VPKINIT[N], where N is the word number (1-16) of this packet.

The VMFIB must contain the values returned by the procedure VOPEN or VOPENDSET. The access path specified by the VACCESSREF must exist, i.e., the user must have previously called one of the VREAD procedure. This procedure will call the REX routine PUT#.

VTRANSFER transfers data from a source device or data set to a destination device or data set and is unprivileged. The procedure definition is VTRANSFER (var VMFIBS: vfib; var VMACBS: vacb; var VMFIBD: vfib; var VMACBD: vacb; NBYTES: integer; SRCEOFFSET: integer; DESTOFFSET: integer).

For inputs:

VMFIBS is the FIB for the source device or data set that has the following fields set:

1. VCHANINFO as set by the VM ISOR acquire ownership or open data set operation.

VMACBS is the ACB for the source device or data set that was referenced by a previous READ operation and has its fields set as follows:

1. VACCESSREF is the value assigned by VM ISORs to uniquely identify the access path that was set up by the previous READ.

2. VTIMEOUT is the time to be added to the system timeout value for the total transfer operation.

VMFIBD is the FIB for the destination device or data set that has the following fields set:

1. VCHANINFO as set by the VM IOSR acquire ownership or open data set operation.

VMACBD is the ACB for the destination device or data set that was referenced by a previous WRITE operation and has its fields set as follows:

1. VACCESSREF is the value assigned by VM IOSRs to uniquely identify the access path that was set up by the previous Write.

NBYTES is the number of bytes to be transferred.

SRCEOFFSET is the byte offset, into the record that was accessed by a previous READ, where the source data starts.

DESTOFFSET is the byte offset, into the record that was accessed by a previous WRITE, where the destination data starts.

For outputs:

VMACBS will have the following fields set on return:

1. VACBCOMPL is the event completion code returned by REX.

2. VACBRESP is the response returned by the source device or data set handler that will have its fields set as follows:

a. VDEVCC contains the status returned from the device or data set handler.

The FIBs associated with the devices or data sets involved in the transfer must contain the values returned by the preceding VOPEN or VOPENDSET operations. A READ access path must exist for the source and a WRITE access path must exist for the destination. This procedure will call the REX routine TRANSFER#. The Transfer procedure allows a user, in one processor, to move data from a second processor to a third processor without having to bring the data into his own processor.

VCONTROL communicates control information to a device handler and is unprivileged. The procedure definition is VCONTROL (var VMFIB: vfib; CTRLBUFF: vctrlinfo; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following fields set:

1. VCHANINFO as set by the procedure that acquired the device.

2. VACQPKTID as set by the procedure that acquired the device.

CTRLBUFF is a record to be included in the control packet being sent to the device. The record contains device-specific control fields to be defined by the target device.

VMIOFLAGS is a set of flags that specify the selected CONTROL options as follows:

1. VCHANNO is the number of the channel to be used for multiple channel devices. It should be zero for devices with only one channel.

For outputs:

VMFIB will have its fields set as follows:

1. VSYSCOMPL contains the REX event completion code.

2. VDEVCOMPL contains the completion code returned by the device handler.

The FIB must contain the values returned by the procedure that acquired the device. This procedure calls the REX routine CONTROL#.

VSETSUBTYPE alters the subtype of a specific device and is system privileged. The procedure definition is VCONTROL (var VMFIB: vfib; SUBTYPE: vbit16; OPRATRID: vbit16).

For inputs:

VMFIB is the FIB for the device whose subtype is to be altered, it has the following fields set:

1. VCHANINFO as set by the VM ISOR acquire ownership operation.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

SUBTYPE is the subtype to be associated with the device.

OPRATRID is the CRT operator's ID if SUBTYPE is non-zero; otherwise, OPRATRID is zero.

For outputs:

VMFIB will have its fields set as follows:

1. VSYSCOMPL contains the REX event completion code.

2. VDEVCOMPL contains the completion code returned by the device handler.

Upon calling VSETSUBTYPE, the user process will be suspended until a response is received. VSYSCMPL and VDEVCMPL will indicate the result. This procedure sends a set device subtype request packet to SYSDEV.

VDELETEKEY builds an access path to a logical record in an indexed data set to permit deletion of the record and is unprivileged. The procedure definition is VDELETEKEY (var VMFIB: vfib; KEYLENGTH: vbit8; KEYPTR: vaddr).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure OPENDSET.

KEYLENGTH is the length (in bytes) of the record key.

KEYPTR is a pointer to a packed array of characters that contains the name of the record key.

For outputs:

VMFIB will have its fields set as follows:

1. VSYSCMPL contains the REX event completion code.

2. VDEVCMPL contains the completion code returned by the device handler.

The FIB must contain the values returned by the procedure VOPEN_DSET. This procedure calls the REX routine CONTROL#.

VRENAME changes the identity of a logical record in an indexed data set by substituting a new key for its current key. The record itself is not disturbed. The function is unprivileged. The procedure definition is VRENAME (var VMFIB: vfib; var VMACB: vacb; OLDKEYLEN: vbit8; NEWKEYLEN: vbit8; OLDKEYPTR: vaddr; NEWKEYPTR: vaddr).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure OPENDSET.

VMACB will have the following fields set:

1. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

2. VPRIORITY is the priority value to be assigned to the operation.

OLDKEYLEN is the length (in bytes) of the record current key.

NEWKEYLEN is the length (in bytes) of the new key.

OLDKEYPTR is a pointer to a packed array of characters that contains the name of the current key.

NEWKEYPTR is a pointer to a packed array of characters that contains the name of the new key.

For outputs:

VMACB will have its fields set as follows:

1. VACCESSREF contains an access reference value (1–255) that will be used by the VM IOSRs in all subsequent I/O operations related to this access path.

2. VACBCOMPL contains the REX event completion code.

3. VACBRESP is the response packet returned by the data set handler that will have its fields set as follows:

a. VDEVCC contains the error code/group code.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1–16) of this packet.

The FIB must contain the values returned by the procedure VOPENDSET. The user process must not alter any of the values returned in the ACB.

VMOVEWINDOW terminates access to the element currently on view in an indexed subfile or CHAIN data set, writing it to disk if necessary. It also moves the window to permit access of a specified element in the current logical record and is unprivileged. The procedure definition is VMOVEWINDOW (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure OPENDSET.

VMACB will have the following fields set:

1. VACCESSREF as set by the procedure that built the access path to the logical record.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

3. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VMOVEWINDOW options as follows:

1. VMARK is a flag that specifies the units of motion. If the flag is reset, the units of motion are record elements. If the flag is set, the units of motion are marks (applicable to string data sets only).

2. VABSOLUTE is the relative access flag that specifies window motion relative to a specific element (c) which is currently on view in the currently accessed logical record. If the flag is set, motion is relative to the first element of the logical record. If the flag is reset, motion is relative to the current element on view.

3. VNULL is set for null continue (MOVEWINDOW), and is reset for normal MOVEWINDOW operation.

4. VDESTINATION.VDEST is the mark or element number to go to (D).

The window will move to C+D if VMARK, VABSOLUTE equal 0,0; to D if VMARK, VABSOLUTE

equal 0,1, to the Dth mark after C if VMARK, VABSOLUTE equal 1,0 and to the Dth mark if VMARK, VABSOLUTE equal 1,1 where VMARK equal 1 is applicable to chained data sets only. C is the number of the record element which is currently on view in the window, relative to the beginning of the logical record and D is an array [1 . . . 2] of vbit16, representing the mark or element number to go to.

For outputs:

VMACB will have its fields set as follows:

1. VACBCOMPL contains the REX event completion code.

2. VACBRESP is the response returned by the data set handler that will have its fields set as follows:

a. VDEVCC contains the error code/group code.

b. Other fields in this packet are unnamed, but may be accessed by VPKTINIT[N], where N is the word number (1-16) of this packet.

The FIB must contain the values returned by the procedure VOPENDSET. The ACB must contain the values returned by the procedure that built the access path to the logical record.

VINSERTELEM terminates access to the current element in a logical record of an indexed subfile data set, writing it to disk if necessary. It creates a new element in the subfile and places it in a specified position relative to the current element. It makes the new element the current element on view. It is unprivileged. The procedure definition is VINSERTELEM (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VACCESSREF as set by the procedure that built the access path to the logical record.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

3. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VINSERTELEM options as follows:

1. VBEFORE is the flag that specifies the position in the logical record to insert the element. If the flag is set, the, new element will be logically inserted before the current element. If the flag is reset, the new element will be logically inserted after the current element.

For outputs:

VMACB will have its fields set as follows:

1. VACBCOMPL contains the REX event completion code.

2. VACBRESP is the response returned by the data set handler that will have its fields set as follows:

a. VIOSTAT contains the error code/group code.

The FIB must contain the values returned by the procedure OPENDSET. The ACB must contain the values returned by the procedure that built the access path to the current logical record.

VDELETELEM terminates access to the current element in a logical record of an indexed subfile data set. It removes from the subfile a logically contiguous run of elements beginning with the current element. The element on view after the remove operation will be the element immediately before the removed run of elements. The function is unprivileged. The procedure definition is VDELETELEM (var VMFIB: vfib; var VCACB: vacd; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VOPENDSET.

VMACB will have the following fields set:

1. VACCESSREF as set by the procedure that built the access path to the logical record.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

3. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS will have the following fields set:

1. ELEMNO is the element number (zero relative to the start of the subfile) of the first element in the run of contiguous elements to be removed.

2. NELEMENTS is the count of the number of contiguous elements to be removed.

For outputs:

VMACB will have its fields set as follows:

1. VACBCOMPL contains the REX event completion code.

2. VACBRESP is the response returned by the data set handler that will have its fields set up as follows:

a. VIOSTAT contains the error code/group code.

The FIB must contain the values returned by the procedure VOPENDSET. The ACB must contain the values returned by the procedure that built the access path to the current logical record.

VMOVELEM move a logically contiguous run of elements in a logical record of an indexed subfile data set to a specified position relative to the current element on view. The function is unprivileged. The procedure definition is VMOVELEM (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following field set:

1. VCHANINFO[O].VCHANRF as set by the procedure VPENDSET.

VMACB will have the following fields set:

1. VACCESSREF as set by the procedure that built the access path to the logical record.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

3. VPRIORITY is the priority value to be assigned to the operation.

VMIOFLAGS is a set of flags that specify the selected VMOVELEM options as follows:

1. VBEFORE is the flag that specifies the position in the logical record to move the run of elements. If the flag is set, the run of elements is move to immediately before the current element on view. If the flag is reset, the run of elements is moved to immediately after the current element on view. The window remains over the same logical element number whick will either be the element that was on view before the move (if VBEFORE is reset) or the former element as specified by VELEMNO (if VBEFORE is set).

ELEMNO is the element number (zero relative to the start of the subfile) of the first element in the run of contiguous elements to be moved.

NELEMENTS is the count of the number of elements to be moved.

For outputs:

VMACB will have its fields set as follows:

1. VACBCOMPL contains the REX event completion code.

2. VACBRESP is the response returned by the data set handler that will have its fields set as follows:

1. VIOSTAT contains the error code/group code.

The FIB must contain the values returned by the procedure VOPENDSET. The ACB must contain the values returned by the procedure that built the access path to the current logical record.

VENDIO terminates a data transfer operation (get, put, or transfer) and tear down the associated access path to a logical data record and is unprivileged. The procedure definition is VENDIO (var VMFIB: vfib; var VMACB: vacb; VMIOFLAGS: vioflags).

For inputs:

VMFIB will have the following fields set:

1. VCHANREF **1** as set by the VM IOSR acquire ownership or OPEN data set operation.

VMACB will have the following fields set:

1. VACCESSREF is the value assigned by VM IOSRs to uniquely identify the access path that is related to this access.

2. VTIMEOUT is the time to be added to the system value to arrive at the timeout interval for the operation.

3. VPRIORITY is the priority value to be assigned to the endio operation.

4. VMIOFLAGS is a set of flags that specify the selected ENDIO options as follows:

a. VABORTACCESS is set when the request is to abort the current access.

For outputs:

VMACB will have the following fields set on return:

1. VACBCOMPL is the event completion code returned by REX.

2. VACBRESP is the response returned by the device or data set handler that will have its fields set as follows:

a. VIOSTAT contains the status returned from the device or data set handler.

The device or data set specified by the FIB must be open. The access path specified by the VACCESSREF must exist, i.e., the user must have previously called one of the access procedures. This procedure calls the REX routine ENDIO#.

VCLOSE closes a logical path to a device or data set and is unprivileged. The procedure definition VCLOSE (var VMFIB: vfib; VMIOFLAGS: vioflags).

For inputs:

VMFIB must have the following fields set:

1. VCHANRF as set by the VM IOSR open.

2. VFDEVNAM as set by the VM IOSR open.

3. VACQPKTID, if applicable, as used by the VM IOSR open.

4. VTIMEOUT is the time (in seconds) to be added to the system timeout value to arrive at the timeout interval for the operation.

5. VPRIORITY is the priority value to be assigned to the close operation.

VMIOFLAGS is set as follows:

1. VCHANNO is the channel number to be closed. This is always zero for noncomplex devices.

For outputs:

VMFIB will have the following fields returned:

1. VSYSCOMPL is the REX event completion code.

2. VDEVCOMPL is the close device or data set completion code.

3. VDEVSTATUS is set to VAQUIRED.

4. VCHANRF for this device is set to zero.

The device or data set specified by the FIB must be open. All accesses to the device or data set must have been terminated. After the device is closed, it must be released. This procedure calls the REX routine CLOSE#. A device or data set is considered closed even if the request completes with errors.

VRLSEDEVICE releases ownership of a system device and is unprivileged. The procedure definition is VRLSEDEVICE (var VMFIB: vfib; VMIOFLAGS: vioflags; APPERRCNT: vbit8).

For inputs:

VMFIB will have the following fields set:

1. VCHANINFO contains the system device name that was established at acquire time and uniquely identifies the device being released.

2. VTIMEOUT is the time to be added to the system timeout value to arrive at the timeout interval for the operation.

VMIOFLAGS is a set of flags that specify the options to the release request as follows:

1. VDOWN set if the release is to include maring the device as "down".

APPERRCNT is the count of the number of device errors detected by the application.

For outputs:

VMFIB will have the following fields set:

1. VSYCOMPL is the REX event completion code.

2. VDEVCOMPL is the release completion code.

3. VDEVSTATUS is set to VRLSED if the operation was successful.

No further communication is possible between the device and the old owner unless the device is required. This procedure sends a release device request packet to SYSDEV.

A set of utility procedures and functions offered by the virtual machine monitor are defined below. The utility procedures and functions which the virtual machine provides include:

1. VLOG_IT print the contents of a packet on the local AEB console.

2. VGET_CREATE PKT return the create packet to the caller.

VLOG_IT transfers a log packet to the system log processes and is unprivileged. The procedure definition is VLOG_IT (LOGFMT: VLOGFORMAT, LOGPKT: VPACKET).

For inputs:

LOGFMT is a record specifying the format of the packet to be logged. LOGFMT must be set as follows:

1. USERFORMAT—user formatted.

2. ASCII—ASCII.

3. BADPKT—bad packet.

4. HEXPKT—HEX. LOGPKT is a 16-word packet to be logged.

If the packet is user formatted, the following fields in the packet must be set:

1. LGUSERFLAG is the user flags (currently unused).

2. LGMSGTYPE is the log message type number that is used by SYSMON for display control.

3. LGMSGNO is the system error group/error code for this packet.

4. LGDATA is the user data to be logged.

If the packet is ASCII or HEX, the packet field VPKTINIT[3] through VPKTINIT[16] contains the user data to be logged.

If the packet is a bad packet, the user should not alter the packet headers. Other fields may be altered at the option of the user.

This procedure calls the REX routine LOGIT, LOGIS$A, LOGIT$B OR LOGIT$H, depending on the value of LOGFMT.

VGETCREATEPKT returns the create process request packet to the caller and is unprivileged. The procedure definition is VGETCREATEPKT (var CREATEPKT: vpacket).

For outputs:

CREATEPKT is returned with the contents of the packet that was sent to cause creation of the calling process.

The following files give a standard Pascal interface to VMM system routines. They are named as follows:

1. VM.CONST—this file contains the constant declarations as well as the keyword CONST. Therefore, it should be included in front of the programs' constant declarations.

2. VM.TYPE—this file contains the type declarations as well as the keyword TYPE. Therefore, it should be included in front of the programs' type declarations.

3. VM.SERVICE—this file contains the code necessary to interface with the VMM procedures. It should be included before any references to VMM procedures.

The REX event completion codes are:

```
VCOMPOK          = X'0';      (* OK *)
VINOUNSOL        = X'1';      (* Unsolicited packet
                                 received. *)
VIOSOL           = X'2':      (* Solicited packet
                                 received. *)
VSIGUNSOL        = X'3';      (* Unsolicited signal
                                 received. *)
VSIGSOL          = X'4';      (* Solicited signal
                                 received. *)
VTMOUTP          = X'5';      (* Pure timer timed
                                 out. *)
VDELETE          = X'F';      (* Delete process re-
                                 quest from JSAM. *)
VMAXGOODREXCC    = X'FF';
VHDR1REJ         = X'101';
VHDR2REJ         = X'102';    (* Transfer, complete,
                                 header rejected. *)
VABORTREJ        = X'103';    (* Transfer aborted,
                                 header rejected. *)
VOUTOFSEQ        = X'104';    (Data packet transfer-
                                 packet out of sequence. *)
VSPACEERR        = X'105';    (* Data packet transfer-
                                 out of buffer space. *)
VMOVEFAIL        = X'106';    (* Data packet transfer-
                                 byte count error. *)
VTIMEOUT         = X'107';    (* Timeout occurred on
                                 I/O, SIGNAL. *)
VECBINVALID      = X'108';    (* Invalid ECB address
                                 specified. *)
VNORESOURCES     = X'109';
VCANTACCESS      = X'10A';    (* Access path already
                                 active; previous I/O
                                 function not yet com-
                                 pleted (IOSRs). *)
VCALLINVALID     = X'10B';
```

The VMM event completion codes are defined by HEX, (decimal) and group as follows:

60	(24576)	VMM good event codes
61	(24832)	General VMM error codes
62	(25088)	Memory file error codes
63	(25344)	Resource manager error codes
64	(25600	P-code errors
65	(25856)	PDS error codes
66	(26112)	VMM utility error codes

The general constants are:

```
VIOOK            = 0;
VMAXCHAN         = 1;         (* Maximum channel for
                                 complex device, 0 or 1 *)
VMAXDSNAMLEN     = 32;        (* Maximum name length
                                 for data sets *)
VMAXREFELEMENT   = 255;
VMEMFILELIMIT    = 10;        (* Maximum number of
                                 memory files per pro-
                                 cess *)
VPMINDSLTH       = 8;         (* Data set length that
                                 fits in a standard para-
                                 meter list *)
VPLMINMOV        = 19;        (* Minimum amount to be
                                 moved for an IOCB *)
VPLMINKEY        = 8;         (* Key length that fits
                                 in a standard parameter
                                 list *)
VIOCBHDRSIZE     = 3;         (* The header size of the
```

-continued

```
                                    IOCB down to the para-
                                    meter list *)

The IOSR option codes are:
"WHERE" options of GET/PUT:
VDATAFLD            = 0;            (* Data-field of record *)
VMEMFLD             = 1;            (* Memo-field of record *)
VKEYFLD             = 2;            (* Key-field of record *)
Utility data areas: VREADDIRECT/VWRITEDIRECT:
UDAAREA             = 0:            (* Utility data area *)
PCAREA              = 2;            (* Primary control area *)
SCAAREA             = 3;            (* Secondary control area *)
DSR function codes:
VDSRCONTINUE        = 6;            (* Continue function
                                    code *)
VDSRDELETE          = 7;            (* Delete function code *)
VDSRRENAME          = 8;            (* Rename function code *)
VDSRUNLOCK          = 9;            (* Unlock function code *)
VDSRMARK            = 13;           (* Mark function code *)

The memory management constants are:
VMAXMAP             = 253;          (* Maximum map segment
                                    number *)

TYPE:
vbit1               = 0..1;
vbit2               = 0..3;
vbit3               = 0..7;
vbit4               = 0..15;
vbit5               = 0..31;
vbit6               = 0..63;
vbit7               = 0..127;
vbit8               = 0..255;
vbit9               = 0..511;
vbit10              = 0..1023;
vbit11              = 0..2047;
vbit12              = 0..4095;
vbit13              = 0..8191;
vbit14              = 0..16383;
vbit15              = 0..32767;
vbit16              = integer;      (* -32768..32767 *)

The general types are:
vaddr               = pointer (integer);
vappclasses         = (vspmcode, vpsodemain, vpcodesub);
vchanrange          = 0..vmaxchan;
vclassrange         = vbit3;
vcomplcode          = packed record
                      case integer of
                      1 : (ver-     : vbit8;
                           group
                           verrcode : vbit8);
                      2 : (verrword : vbit16)
end (* vcomplcode *);
vdevcompcode        = packed record
                      vioerrgroup   : vbit8;
                      vioerrcode    : vbit8;
end (* vdevcompcode *);
vecbid              = vbit16;
vfcrange            = vbit4;
vlogformat          = (userformat, ascii, badpkt, hexpkt);
vlonginteger        = array [1..2] of vbit16;
vmaprange           = 0 . . vmaxmap;  (* number of segment maps
                                      in GPP *)
vmfrname            = packed array    [1..8] of char;
vpageindexrange     = 1..6;           (* number of pages per
                                      segment map *)
vprogid             = packed record
                      fill1         : vbit16;
                      progno        : vbit16;
                      fill2         : vbit16;
end (* vprogid *);
vprogramname        = packed array [1 . . 12] of char;
                      (* .CODE is added by CXP *)
vrankrange          = vbit8;
vpriority           = packed record
                      class         : vclassrange;
                      fil1          : vbit1;
                      rank          : vrankrange;
end (* vpriority *);
vsdtype             = (vacq, vopid, vrls, vssub);
                      (* type of call to SYSDEV *)
```

-continued

```
vstypelist          =  packed array [1..14] of vbit8;
                       (* subtype list for acquire *)
vuserrefval         =  vbit16;

The declarefc definitions are:
vfcs                =  (VFC1,VFC1,VFC2,VFC3,VFC4,VFC5,VFC6,
                       VFC7,VFC8,VFC9,VFCA,VFCB,VFCC,VFCD,
                       VFCE,VFCF);
vfcset              =  set pf vfcs'

The packet definitions are:
vbidrange           =  vbit7;
vcxidrange          =  vbit5;
vcxsizerange        =  vbit4;
vjobnorange         =  vbit16;  (* 0..65535 *)
vmemrange           =  vbit15;  (* 0..65535 *)
vprtyrange          =  vbit12;
vspidrange          =  vbit8;
vipidrange          =  vbit13;  (* vcxid = 5 bits,
                                   vspid = 8 bits *)
vpid                =  packed record
                       fil0          :  vbit4;
                       cxid          :  vcxidrange;
                       bid           :  vbidrange;
                       fil1          :  vbit8;
                       spid          :  vspidrange;
end (* vpid *);
vloadstates         =  (load, loading, loaded);
vpgmreqmap          =  packed array [1..14] of vbit4;
vpgmrspmap          =  packed array [1..14] of vmaprange;
vpacket             =  packed record
case integer of
 1: (vfcode         :  vfrange;
     vcxid          :  vcxidrange;
     vbid           :  vbidrange;
     vrspid         :  vspidrange;
     vspid          :  vspidrange;
     vrcode         :  vfcrange;
     vrbid          :  vbidrange;
     vipkt          :  array [1..13] of vbit16);
 2: (vpktinit       :  array [1..16] of vbit16);
 3: ((* I/O return packet definitions *)
     fil1           :  array [1..5] of vbit16;
     vdevcc         :  vcomplcode;
     fil2           :  array [1..3]3] of vbit16;
     vdatalen       :  vbit16 );
 4: ((* CR = Create Process request packet *)
     crfil0         :  array [1..3] of vbit16;   (* words 0-2 *)
     crjobno        :  vbit16;                   (* word 3 *)
     crfil7         :  boolean;                  (* word 4 *)
     crbypass       :  boolean;
     crspecial      :  boolean;
     crfil4         :  vbit1;
     crclass        :  vclassrnage;
     crfil6         :  vbit1;
     crrank         :  vrankrange;
     crprogid       :  vprogid;                  (* words 5-7 *)
     crfil1         :  vbit3;                    (* word 8 *)
     crcxno         :  vcxidrange;
     crfil2         :  vbit4;
     crcxsize       :  vbit4;
     crfil3         :  vbit13;                   (* word 9 *)
     crnonshare     :  boolean;
     crappclass     :  vappclasses;
     crfil5         :  array [1..2] of vbit16;   (* words A&B *)
     crjobparms     :  array [1..4] of vbit16)   (* words C&F *)
 5: ((* CRM         =  create packet sent by resource manager
                       to virtual machine monitor for new
                       process. Also sent back to resource
                       manager to give load status and sent
                       from process to resource manager on
                       deallocation. *)
     crmfil0        :  array [1..3] of vbit16;   (* word 0-2 *)
     crmprogname    :  vprogrammame;             (* word 3-8 *)
     crmfil1        :  vbit8;                    (* word 9 *)
     crmdataseg     :  vmaprange;
     crmfil2        :  integer;                  (* word A *)
     cmrfil3        :  vbit2;                    (* word B *)
     crmpds         :  boolean;                  (* 0 = Delphi.
                                                    driver *)
                                                 (* 1 = Delphi.
                                                    pascal *)
```

-continued

```
    crmmemavail          :  boolean;
    crmjsamcreated       :  boolean;
    crmrpttocreator      :  boolean                 (* True if process
                                                    was created thru
                                                    FC 'B' *)
    crmprogid            :  vprogid;                (* word C-E : pro-
                                                    gram ID from create
                                                    packet *)
    crmabortchr          :  integer);               (* word F : the
                                                    abort character
                                                    from PSS *)
 6: ((* MF0             =  Memory File Open request packet, also
                            used for extended memory files *)
    mfofil0              :  array [1..3] of vbit16; (* word 0-2 *)
    mfoname              :  vmfname;                (* word 3-6 *)
    mfoinitrcdcnt        :  integer;                (* word 7 *)
    mfomaxrcdcnt         :  integer;                (* word 8 *)
    mfopagesperrcd       :  integer;                (* word 9 *)
    mfodatasegno         :  vmaprange;              (* word A *)
    mforelpgidx          :  vpageindexrange;        (* word B *)
    mfextendrcucnt       :  integer;                (* word C *)
 7: ((* MFR             =  Memory File Response packet and Memory
                            File Close Request packet *)
    mfrfil0              :  array [1..3] of vbit16; (*word 0-2 *)
    mfrstatus            :  vcomplcode;             (* word 3 *)
    mfrno                :  vbit16;                 (* word 4 *)
    mfrfil1              :  array [1..2] of integer; (* word 5-6 *)
    mfrcrntrcdcnt        :  vbit16                  (* word 7 *)
    mfrfil2              :  array [1..4] of integer; (*word 8-B *)
    mfrcdno              :  integer);               (* word C *)
 8: ((* CRN             =  Create Node Request packet *)
    crnfil0              :  array [1..3] of vbit16; (* word 0-2 *)
    crnjobno             :  vbit16;                 (* word 3 *)
    crnfil1              :  vbit16;                 (* word 4 *)
    crnnid               :  boolean;                (* word 5 *)
    crnprim              :  boolean;
    crnrecov             :  boolean;
    crnfil2              :  boolean;
    crnloss              :  vbit4;
    crnfil3              :  vbit6;
    crnct                :  vbit2;
    crnpgm1              :  vprogid;                (* word 608 *)
    crnpgm2              :  vprogid;                (* word 9-B *)
    crnpgm3              :  vprogid;                (* word C-E *)
    crnfil4              :  vbit9;                  (* word F *)
    crnuserbid           :  vbidrange);
 9: ((* CNR             =  Create Node Response packet *)
    cnrfil0              ;  array [1..3] of vbit16; (*word 0-2 *)
    cnrjobno             :  vbit16;                 (* word 3 *)
    cnrpgm               :  vbit16;                 (* word 4 *)
    cnrfil1              :  vbit15;                 (* word 5 *)
    cnrnakflag           :  boolean;
    cnrnpid1             :  vpid;                   (* word 6-7 *)
    cnrnpid2             :  vpid;                   (* word 8-9 *)
    cnrnpid3             :  vpid;                   (* word A-B *)
    cnrnpid4             :  vpid;                   (* word C-D *)
    cnrnpid5             :  vpid);                  (* word E-F *)
10: ((* LG = Log packet *)
    lgfil1               :  integer;                (* word 0 *)
    lgfil2               :  vbit13;                 (* word 1 *)
    lguserflag           :  vbit3;
    lgmsgtype            :  vbit15;                 (* word 2 *)
    lgfil13              :  packed array (3..7] of vbit16; (*word 307 *)
    lgmsgno              :  integer;                (* word 8 *)
    lgdata               :  packed array [1..7] of vbit16); (* word 9-F *)
11: ((* CPR             =  Create Process Response packet *)
    cprhdr               :  packed array [0..2] of vbit16; (* word 0-2 *)
    cprjobno             :  integer;                (* word 3 *)
    cprfil4              :  vbit15;                 (* word 4 *)
    cprpgmload           :  boolean;
    cprerrno             :  vcomplcode;             (* word 5 *)
    cprpgm               :  vprogid;                (* word 6-8 *)
    cprfil               :  packed array [9..15] of vbit16); (* word 9-F *)
12: ((* LRS             =  Program Load Response packet *)
    lrshdr               :  array [0..2] of vbit16: (* word 0-2 *)
    lrstatus             :  vloadstates;            (* word 3 *)
    lrserror             :  boolean;
    lrsmap               :  vpgmrspmap);            (* word 4-A *)
13: ((* LRQ             =  Program Load Request packet *)
    lrqhdr               :  array [0..2] of vbit16; (* word 0-2 *)
    lrqfil0              :  vbit2;                  (* word 3 *)
    lrq                  :  (vloaded, vres, vnotifyl, vnotifyd);
```

-continued

```
    lrqnumofmaps        :   vbit4;
    lrqdseg             :   vmaprange;
    lrqdsnum            :   integer;            (* word 4 *)
case integer of
  1: (lrqpgmid          :   vprogramname;       (* word 5-A *)
    lrqprogno           :   integer);           (* word B *)
  2: (lrqmap            :   vpgmreqmap;         (* word 5-8 *)
    lrqsegjmap          :   vpgmreqmap));       (* word 9-C *)
end (* vpacket *);
```

The packet definition for the acquire, release and set subtype packets sent to SYSDEV are:

```
vaqupacket              =   packed record
case integer of
  1: (vpkt              :   vpacket);
  2: (fil1              :   packed array [1..3] of vbit16;
                            (* acquire *)
    vjobno              :   vbit16;
    vusrref             :   vbit16;
    vqopt               :   vbit1;
    vacqtype            :   vbit1;
    fil2                :   vbit2;
    vprior              :   vbit4;
    vopcode             :   vbit8;
    vintnm              :   vbit16;
    vusnfc              :   vfcrange;
    vusncx              :   vcxidrange;
    vusnbid             :   vbidrange;
    vdevtype            :   vbit8;
    vusnspid            :   vbit8;
    vstypelst           :   bstypelist);
  3: (fil3              :   array [1..5] of vbit16;   (* release *)
    vapperrcnt          :   vbit8);
  4: (fil4              :   array 1..7] of vbit16;    (* set sub-
                                                        type *)
    vsubtype            :   vbit16;
    vzenopid            :   vbit16);
end (* vacqpacket *);
```

The VCOM definition is:

```
vmfrcd                  =   packed record           (* Each entry in the array of
                                                       memory files *)
mfname                  :   vmfrname;
mfno                    :   integer;                (* Internal number assigned to
                                                       file *)
mfcurrcd                :   vbit11;                 (* Current record that is
                                                       installed *)
mfrelmempy              :   vbit5;                  (* Relative page in this seg-
                                                       ment where the file starts
                                                       (1..16) *)
end (* vmfrcd *);
vcomrec                 =   record                  (* Record kept in VCOM to
                                                       describe the process for
                                                       VMM. *)
vccrpkt                 :   vpacket;                (* The create process request
                                                       packet sent by the resource
                                                       manager to the virtual
                                                       machine monitor to create the
                                                       process *)
vcrmpkt                 :   vpacket;                (* The create request communi-
                                                       cation packet for this pro-
                                                       cess as defined by the re-
                                                       source manager. *)
vmftable                ;   array [1..vememfilelimit]
                            of fmvrcd;
end (* vcomrex *);
vcomrptr                =   pointer (vcomrec);
```

The SELFPID definition is:

```
vipidrec                =   packed record
vfiller3                :   vbit3;
vpidcx                  :   vcxidrange;
vpidspid                :   vspidrange
end (* vipidrec *);
vpidrec                 =   packed record
vpidbid                 :   vbidrange;
vipid                   :   vipidrec
end (* vpidrec *);
```

The time management types are:

```
vmmtimeunits            =   (vtmnull, vsecs, vmills);
vdtrectype              =   packed record
```

-continued

```
year                    :   0..99;
month                   :   1..12;
day                     :   1..31;
doy                     :   1..366;
dow                     :   1..7;
hour                    :   0..23;
min                     :   0..59;
sec                     :   0..59;
hsec                    :   0..99;
end (* vdtrectype *);

The send / signal definitions are:
vcsfselements           =   (vcwf0,fcwf1,vdwf2,fcwf3,fcwf4,fcwfT,
                            vcwf6,fcwf7,fcwfL,vcwfF,vdwfH,vdwfA,
                            vcwfE,vcwfS,vcwfR,vcwfN);
vmemtypes               =   (VSCRATCHPAD, VMNULL, VDATAMEN);
vaglist                 =   packed record
case integer of
  1: ((* Actual definition of SEND argument list *)
     VALRESECBADR       :   vbit16;          (* =0 *)
     VALUSERREFVAL      :   vbit16;          (* response *)
  2: ((* Definition to initialize argument list
     to zero *)
     VALFINIT           :   array [0..7] of vbit16);
  3: ((* Definition of STARTTIMER argument list *)
     C1                 :   vbit5:
     PUNITS             :   vmmtimeunits;    (*minutes or seconds *)
     C2                 :   vbit4;
     PECBCODE           :   vbit5;           (* greater than or =
                                             to 16 *)
     PUREFID            :   vuserrefval;     (* user reference value *)
     PEXADDR            :   integer;         (* SPM set *)
     PTMIPID            :   vipidrec;        (* SELFPID *)
     PTIMELEN           :   integer);        (* number of units *)
  4: ((* Definition of DATETIME argument list *)
     PYEAR              :   integer;         (* year *) (*word 0 *)
     F1                 :   vbit4;           (* filler *) (* word 1 *)
     PMONTH             :   1..12;           (* month - 4 bits *)
     F2                 :   vbit3;.          (* filler *)
     PDAY               :   1..31;           (* day of month-5 bits*)
     F3                 :   vbit4;           (* filler *) (* Word 2 *)
     PDOY               :   1..366;          (* day of year-9 bits *)
     PDOW               :   1..7;            (* day of week-3 bits *)
     F4                 :   vbit3;           (* filler *) (* word 3 *)
     PHOUR              :   0..23;           (* hour (military) -
                                             5 bits *)
     F5                 :   vbit2;           (* filler *)
     PMIN               :   0..59;           (* minute of hour - 6
                                             bits *)
     F6                 :   vbit2;           (* filler *) (* word 4 *)
     PSEC               :   0..59;           (* second of minute -
                                             6 bits *)
     F7                 :   vbit1;           (* filler *)
     PCSEC              :   0..99;           (* hundredths of seconds -
                                             7 bits *)
end (* varglist *);
VCRTLWRD                =   packed record
Case integer of
  1: ((* Actual Definition of SEND / SIGNAL Control
     Word *)
     VCWN               :   boolean;         (* Formatted ECB for
                                             notify supplied *)
     VCWR               :   boolean;         (* Formatted ECB for resp.
                                             supplied *)
     VCWS               :   boolean          (* Switch header before
                                             transfer *)
     VCWE               :   boolean;         (* 1=wait specific;
                                             U=wait any *)
     VCWA               :   boolean;         (* 0=default to L & L+1;
                                             1=don't *)
     VCWH               :   boolean;         (* Create return header *)
     VCWF               :   boolean;         (* Free packet to avail-
                                             able space *)
     VCWL               :   boolean;         (* 1=long format; 0=short
                                             format *)
     VCWM               :   vmemtypes;       (* 0=scratchpad; 10=data
                                             memory *)
     VCWT               :   boolean;         (* Time response *)
     VCW4               :   boolean;
     VCWC               :   vfcrange);       (* function code for return
                                             hdr/resp *)
  2: P ((* Definition to initialize Control Word to
```

-continued

```
    zero *)
    VCWFINIT          :   vbit16);
  3: ((* Definition to act on Ctron word as a set *)
    VCWFSET           :   set of vcwfselements)
end (* vctrlwrd *);

The wait / check_event definitions are:
vchecktype            =   ( VCHKANY, VCHKUSERREF, VCHKRESPONSEID );
vwaittype             =   ( VWAITANY, VWAITSPECIFIC );

The I/O definitions are:
chanreft              =   0..255;
volid                 =   packed record
case integer of
  1: (volidmov        :   array [i..3] of Vbit16;
  2: (volidinit       :   packed array [1..6] of char)
end (* volid *);
vdatasetinfo          =   packed record

case integer of
  1: ((* used to move DSINFO into REXPARMLIST *)
    vdsimov           :   array [1..30] of integer;
  2: (
    vdsexcl           :   boolean;          (* 1= exclusive open,
                                               0= shared open *)
    vdsrdwr           :   boolean;          (* 1= write, 0= read *)
    vdsutil           :   boolean;          (* 1= utility open,
                                               0= normal open *)
    fil1              :   vbit13;
    vdsuncat          :   boolean;          (* 1= uncatalogued,
                                               0= catalogued *)
    vdsup             :   boolean;          (* 1= duplicated,
                                               0= not duped *)
    fil2              :   vbit14;
    fil3              :   vbit16;
    fil4              :   vbit8;
    vparmlth          :   vbit8;            (* parameter list length *)
    vuser             :   packed array [1..5] of vbit8;
                                            (* requester ID *)
    vpart             :   vbit8;            (* part number (tape
                                               reel) *)
    vdsnamlen         :   vbit8;            (* data set name length *)
    vdsnam            :   packed array [1..32] of char;
                                            (* data set name *)
    volid1            :   volid;            (* first volume ID *)
    volid2            :   volid;            (* second volume ID *)
end (* vdatasetinfo *);
vdevname              =   vbit16;
viofunccodes          =   (vfcopen, vfcclose, vfccontrol,
                           vfcaccess, vfcendio, vfcget, vfcput,
                           vfctransfer, vfcopendev, vfcaccdev,
                           vfccontinue, vfccntrlds, vfcgdata);
vrefrange             =   1..255;
vrefset               =   set of vrefrange;
vreftype              =   (vrefaccess, vrefchan);
vdvstatcode           =   (vaquired, vopened, vrlsed)
vrecordidtype         =   array [1..3] of vbit16;
(* Access conrrol block definition *)
vacb                  =   packed record
VACCESSREF            :   vbit8;            (* VM IOSR assigned
                                               reference value. *)
VACBCOMPL             :   vcomplcode;       (* REX event completion
                                               code. *)
VACBRESP              :   vpacket;          (* Rtn packet from opera-
                                               tion (13 wds) *)
VTIMEOUT              :   vbit8;            (* User defined *)
VPRIORITY             :   vbit8;            (* User defined *)
VACCHANNO             :   vchanrange;       (* Index into FIB table
                                               of channels *)
end (* vacb *);

The file information block definition is:
vfig                  =   packed record
VCHANCNT              :   vbit8;            (* No. of channels sup-
                                               ported for FIB *)
VCHANINFO             :   array [vchanrange] of
                          packed record
VCHANRF               :   vchanreft;        (* VIOSR assigned channel
                                               reference value *)
VFDEVNAM              :   vdename;          (* Device name for
                                               channel *)
VCQSBID               :   vbidrange;        (* BID of channel *)
```

-continued

```
VCHDEVTYP          :  vbit8;              (* Device type of device
                                             for this channel *)
end;
VOPID              :  vbit16;             (* Operator ID from
                                             acquire response *)
VSYSCMPL           :  vcomplcode;         (* REX event completion
                                             code. *)
VDEVCMPL           :  vcomplcode;         (* DEvice/ data set
                                             completion code. *)
VACQPKTID          :  vbit16;             (* Acquire Packet ID *)
VOPENRESP          :  vpacket;            (* Open response packet
                                             for use in record and
                                             playback *)
VTIMEOUT           :  vbit8;              (* User defined. *)
VPRIORITY          :  vbit8;              (* User defined. *)
VDEVSTATUS         :  vdvstatcode;        (* Device status -
                                             ACQUIRED or OPENED *)
end (* vfib *);

The I/O flag definitions are:
vioflags           =  packed record
VACQQUEUE          :  boolean;            (* SEt if SYSDEV permitted
                                             to queue acquire *)
VACQSEIZE          :  boolean;            (* Set if dev chanl to be
                                             seized during acq *)
VDOWN              :  boolean;            (* True if release and
                                             down required *)
VDIALOUT           :  boolean;            (* True if zeize is for
                                             dialout *)
(* VREADDIRECT, VWRITEDIRECT, VREADKEY *)
VUPDATE            :  boolean;            (* Set if updating read
                                             (i.e. read w/lock *)
(* VREADKEY *)
VGREATER           :  boolean;            (* Used on keyed rcds to
                                             find next grtr key *)
VMEMOREC           :  boolean;            (* Set if memo rcd is not
                                             to be retrieved *)
VDATAREC           :  boolean;            (* Set if data rcd is not
                                             to be retrieved *)
VKAPPROX           :  boolean;            (* Set if read is approx,
                                             used w/VGREATER *)
(* VWRITEKEY *)
VREPLACE           :  boolean;            (* Set if write is update
                                             of existing rcd *)
VELEMSELECT        :  boolean;            (* Used by VALLOCATE -
                                             see spec *)
VUNITS             :  boolean;            (* United of motion for
                                             VMOVEWINDOW *)
VRELATIVE          :  boolean;            (* Window motion rela-
                                             tive to current
                                             record *)
(* VCONTINUE *)
VNULL              :  boolean;
(* VCONTINUE: indexed record *)
VSAMREC            :  boolean;
VNODATA            :  boolean;
(* VMARK, VUNMARK, VCONTINUE: string *)
VMARK              :  boolean;
(* VCONTINUE: subfile *)
VBEFORE            :  boolean;            (* Position in logical
                                             rcd to insert
                                             element *)
(* VENDIO *)
VLOCK              :  boolean;
VABORTACCESS       :  boolean;            (* Abort current
                                             access *)
(* VWGET, VWPUT *)
VPGWHERE           :  vbit2;
(* VWPUT *)
VFIELD             :  boolean;
VCHANNO            :  vchanrange;         (* Set to chanl no.
                                             of multi-chanl
                                             device *)
(* VCONTINUE: string/subfile *)
VABSOLUTE          :  boolean;
VDESTINATION       :  packed record
case integer of
  1: ( VDESTINIT   :  array [1..3] of vbit16);
  2: ( VDEST       :  array [1..2] of vbit16
     VELEMENT      :  vbit16)
end (* VDESTINATION *)
(* VCONTROL *)
```

-continued

```
VFCSPECIFIED        :  boolean;              (* True if VCNTRLFC
                                                is specified *)
VCNTRLFC            :  vfcrange;             (* Function code to use
                                                in VCONTROL if
                                                VFCSPECIFIED is
                                                true *)

The REX I/O control word definitions are:
VIOWSELEMENTS       =  (VIOCWS0, VIOCWS1, VIOCWS2,
                        VIOCWS3, VIOCWSQ, VIOCWST,
                        VIOCWSM, VIOCWSA, VIOCWSW,
                        VIOCWSR, VIOCWS0, VIOCWSF,
                        VIOCW12, VIOCW13, VIOCW14,
                        VIOCW15 );
VIOCWTYPES          =  (VOICWDETAIL, VIOCWINIT,
                        VIOCWSET);
VIOCTRLWRD          =  packed record
case viocwtypes of
VIOCWDETAIL         :  ((* Actual definition of REX I/O
                        control word *)
VIOCWED             :  vfcrange;             (* REX I/O event
                                                code *)
VIOCWF              :  boolean;              (* 1 = Use FC speci-
                                                fied in VIOCWFC
                                                0 = Use REX
                                                defaults *)
VIOCWO              :  boolean;              (* 1 = I/O running
                                                as caller's routine *)
VIOCWR              :  boolean;              (* 0 = Do not return
                                                input parameter
                                                list *)
VIOCWW              :  boolean;              (* 1 = WAIT$E *)
VIOCWA              :  boolean;              (* 0 = Do not invoke
                                                ENDIO; pertains to
                                                ACCESS, CONTINUE, GET,
                                                PUT, TRANSFER; on
                                                CONTROL & OPEN:
                                                0 = chanl 0 & 1 =
                                                chanl 1 *)
VIOCWM              :  boolean;              (* For OPEN & CONTROL,
                                                loc of acquire resp
                                                packet: 0=scratchpad,
                                                1=data memory. For
                                                ACCESS & CONTINUE,
                                                0=input, 1=output *)
VIOCWT              :  boolean;              (* 1 = Time response *)
VIOCWQ              :  boolean;              (* 1 = End of sequence *)
VIOCWFC             :  vfcrange);            (* REX I/O function
                                                code *)
VIOCWINIT           :  ((* Definition to initialize I/O
                        control word to zero *)
VIOCWINIT           :  vbit16;
VIOWSET             :  ((* Definition to act on I/O con-
                        trol word as a set *)
VCWFSET             :  set of viocwselements)
end (* vioctrlwrd *);

The IOSR REX parameter list control word formats are:
VIODCRTL            =  packed record
case integer of
 1: (                                        (* VOPEN DSET *)
    VIODINIT        :  vbit16);
 2: (                                        (* VREADDIRECT, VREDMAPPED *)
    FIL2            :  vbit13;
    VIODUTIL        :  vbit2;
    VIODUPD         :  boolean);
 3: (                                        (* VWGET, VWPUT *)
    FIL3            :  vbit3;
    VIODFLD         :  vbit2;
    FIL4            :  vbit10;
    FIELD           :  boolean);
 4: (                                        (* VREADKEY *)
    FIL5            :  vbit11;
    NODATA          :  boolean;
    FIL55           :  vbit1;
    APPROX          :  boolean
    GREATER         :  boolean;
    UPDATE          :  boolean);
 5: (                                        (* VENDIO *)
    FIL6            :  vbit1;
    ABORT           :  boolean;
    LOCK            :  boolean;
```

-continued

```
    FIL7                :   vbit13;
 6: (                       (* VMARK, VUNMARK *)
    FIL8                :   vbit15;
    UNMARK              :   boolean);
 7: (                       (* VLOCK, VUNLOCK *)
    FIL9                :   vbit15;
    VUNLOCKALL          :   boolean);
 8: (                       (* VCONTINUE *)
    FIL10               :   vbit1;
    ISTRING             :   boolean;
    FIL11               :   vbit7;
    INULL               :   boolean;
    SAMEREC             :   boolean;
    INODATA             :   boolean;
    FIL13               :   vbit4;
 9: (                       (* VPCONTIUE, VMOVEWINDOW,
                            VINSERTELEM, VDELETELEM,
                            VMOVELEM *)
    FIL14               :   vbit1;
    STRING              :   boolean;
    FIL15               :   vbit7;
    NULL                :   boolean;
    MARK                :   boolean;
    ABSOLUTE            :   boolean;
    BEFORE              :   boolean;
    FIL16               :   vb1';
    OPERATION           :   vbit2);
10: (                       (* WRITEKEY *)
    FIL17               :   vbit9;
    REWRITE             :   boolean;
    FIL18               :   vbit6);
end (* viodcrtl *);

The VNM IOCB definition is:
VCRTRDCRTL          =   packed record
VSOM                :   vbit8;          (* start of message
                                           character *)
VPARTNO             :   vbit8;          (* partition number *)
VLINEITEM           :   vbit8;          (* line number *)
VEOM                :   vbit8;          (* end of message
                                           character *)
end (* vcrtrdcrtl *);
VCRTLINFO           =   packed array    [1..18' of vbit8;
                                        (* control information *)
VDEVTYPE            =   vbit8;          (* device type for
                                           acquire *)
(* $P *)

The REX I/O parameter list is:
VREXIOPARMLIST      =   packed record
case integer of
  1: ((* Actual definition of REX I/O parameter
     list *)
FIL1                :   array ('..2]    of vbit16; (* word 0-1 *)
                                        (* Primary & secondary
                                        rtn address *)
VPLPARMID           :   vbit16;         (* Acquire packet address
                                        or I/O list address *)
                                        (* word 2 *)
VPLCHANREF          :   vchanreft;      (* Chanl no. for access *)
                                        (* word 3 *)
VPLACCESSREF        :   vbit8;          (* Access reference
                                        value *)
VPLOPCTRL           :   viodctrl;       (* Operation control *)
                                        (* word 4 *)
FIL2                :   vbit8;          (* word 5 *)
VPLPRIORITY         :   vbit8;          (* Priority *)
FIL3                :   vbit16;         (* word 6 *)
FIL4                :   vbit8;          (* word 7 *)
VPLPSTLTH           :   vbit8;          (* Access key or para-
                                        meter list lth *)
VPLPLSTPTR          :   vaddr);         (* Ptr to data key *)
                                        (* word 8 *)
  2: ((* Initialization definition of REX I/O para-
     meter list *)
     VPLINIT        :   array [;..16] of integer):
                                        (* word 0-F *)
  3: ((* Definition for disk direct access *)
     FIL6           :   array [1..4] of vbit16;  (* word 0-3 *)
     FIL7           :   vbit15;          (* word 4 *)
     VPLUFLAG       :   boolean;         (* Update flag *)
     FIL8           :   array [1..3] of vbit16;  (* word 5-7 *)
```

-continued

```
    VPLELEMNO      :  vlonginteger;        (* word 8 *)
    VNUMBLKS       :  vbit16);             (* Anticipate buffering *)
                                           (* word 9 *)
35: ((* Definition for VMARK & VUNMARK *)
    FIL9           :  array [1..8] of vbit16;
    VMARK          :  vbit16;              (* Mark number *)
                                           (* word 8 *)
    VOFFSET        :  vbit16;              (* Mark offset in elem. *)
                                           (* word 9 *)
    VREFVAL        :  vbit16;              (* User reference value *)
                                           (* word A *)
 4: ((* Definitions for VTRANSFER, VCONTROL, VREADCRT,
    & VWRITEDEV *)
    FILA           :  array [1..6] of vbit16;  (* word 0-5 *)
    VPPDEVIN       :  vdevname;            (* Internal device name *)
                                           (* word 6 *)
case integer of
 1: (FILB          :  array [1..2] of vbit15;  (* word 7-8 *)
    VCNTRL         :  vcrtrdctl);          (* word 9 *)
                                           (* SOM. partition number,
                                           line item, and EOM for
                                           VREADCRT *)
 2: (FILC          :  array of 1..2] of vbit16;
                      (* word 7-8 *)
    VPPWRTBYTCT    :  vbit16);
                      (* # bytes for VWRITEDEV *)
                      (* word 9 *)
 3: (VCNTRLBUF     :  vcrtlinfo):          (* for VCONTROL *)
                                           (* word 7 *)
 4: (VPLCHAND      :  vchanreft;           (* word 7 *)
    VPLACCD        :  vbit8; (* Chanl & access ref for
                      VTRANSFER *)
end (* vrexioparmlist *);
VIOLST             =  array [1..100] of integer;
VIOCB              =  record
VIOCBSIZE          :  integer;
VMOVSIZE           :  integer;
VREXIOTIMEOUT      :  vbit8;
VREXPARMLIST       :  vresioparmlist;
VIOLIST            :  violst;
end (* viocb *);
```

A listing of various programs and subroutines for implementing a specific embodiment of the invention was included in the original application for this patent. This listing was deleted from the application prior to publication but remains in the Patent and Trademark Office file of the application for this patent.

What is claimed is:

1. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets which each include an inbasket portion and an outbasket portion coupled to store and retrieve digital representation of voice messages at each of the plurality of individually addressable message baskets therein; and

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a voice message received over the particular telephone line, and to forward a representation of a voice message stored in the particular message basket to at least one other of the individually addressable message baskets, the control system responding to receipt of data signals over the particular telephone line including a personal identification number associated with an owner of the particular message basket by enabling account activities including retrieval of voice messages from the inbasket of the particular message basket and storage of messages to be forwarded in the outbasket of the particular message basket and blocking account activities until a personal identification number has been received.

2. The automated telephone voice service system according to claim **1** wherein the store stores a representation of a voice prompting message explaining which combinations of data signals actuate particular services provided by the service system and wherein the control system responds to activation of a particular telephone line by retrieving the voice prompting message from the store and communicating the voice prompting message over the particular telephone line.

3. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets which each have an inbasket portion and an outbasket portion, the store being coupled to store and retrieve representations of voice messages at each of the plurality of individually addressable message baskets therein; and

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a

voice message received over the particular telephone line, and to forward a representation of a voice message stored in the particular message basket to at least one other of the individually addressable message baskets, the control system storing in the inbasket portion of the particular message basket representations of voice messages directed to the particular message basket, storing in the outbasket portion of the particular message basket a representation of a voice message that is to be forwarded, and storing in the inbasket of each message basket to which a representation of a voice message is to be forwarded a vector pointer identifying the particular message outbasket and the voice message representation which is to be forwarded thereto.

4. The automated telephone voice service system according to claim 3 wherein the control system returns to a state of a newly activated telephone line upon receipt of data signals over the particular telephone line indicating a change function command.

5. The automated telephone voice service system according to claim 1 wherein at least a subplurality of the predetermined plurality of telephone lines are direct inward dial telephone lines including first and second groups of lines having respectively first and second groups of mutually exclusive telephone numbers associated therewith, with each first and second group of lines being associated with a different common account and with each telephone line of the subplurality having a message basket associated therewith which message basket has an address including a group field identifying the group of telephone lines to which the message basket is associated and an individual field uniquely identifying the message basket within a group of message baskets which are associated with the group of telephone lines.

6. The telephone voice service system according to claim 5 wherein the particular telephone line is one of the subplurality of telephone lines and wherein the particular message basket associated by the control system with the particular telephone line is, in response to data signals defining only an individual field, a message basket identified thereby within the same group as the particular telephone line, and in response to data signals defining both a group field and an individual field, a message basket indicated by the defined individual field within a group indicated by the defined group field.

7. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets, the store being coupled to store and retrieve representations of voice messages at each of the plurality of individually addressable message baskets therein, the store further storing a representation of a voice prompting message explaining which combinations of data signals actuate particular services provided by the service system; and

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a voice message received over the particular telephone line, and to forward a representation of a voice message stored in the particular message basket to at least one other of the individually addressable message baskets, the control system responding to activation of a particular telephone line by retrieving the voice prompting message representation from the store and communicating the voice prompting message over the particular telephone line, the control system operating in an absence of data signals received over the particular line to select as the particular message basket a message basket having a predetermined association with the line, to retrieve from the store and communicate over the line a voice prompting message indicating that a voice message may be received and stored and to store in the particular message basket a representation of any voice message subsequently received over the particular line.

8. The automated telephone voice service system according to claim 1 wherein the data signals are telephone keyboard activated signals defining predetermined commands and data groups and wherein the control system responds to a command only if it is preceded by an ATTENTION signal.

9. The automated telephone voice service system according to claim 8 wherein the control system responds to a data group upon entry of a TERMINATION signal.

10. The automated telephone voice service system according to claim 9 wherein the ATTENTION signal is a tone signal generated by actuating a star key on a standard telephone keyboard and the TERMINATION signal is a tone signal generated by actuating a number sign key on a standard telephone keyboard.

11. The automated telephone voice service system according to claim 10 wherein the control system operates to initiate a predetermined time period upon receipt of the ATTENTION signal from a particular telephone line and responds to a subsequently received command only if a first data signal of the command is received within the predetermined time period.

12. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets, the store being coupled to store and retrieve representations of voice messages at each of the plurality of individually addressable message baskets therein;

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a voice message received over the particular telephone line, and to forward a voice message representation stored in the particular message basket to at least one other of the individually addressable message baskets;

an operator console for providing communication with a human operator and generating operator initiated data signals which are coupled to the control system; and

wherein the control system responds to data signals received over the particular telephone line indicat-

ing that a telephone calling the particular telephone line is a dial type of telephone by coupling the particular telephone line to the operator console and responding to data signals generated by the operator console as if the console generated data signals had been received over the particular telephone line.

13. The automated telephone voice service system according to claim 1 wherein the control system includes a plurality of line interface circuits which are selectively coupled to activated ones of the plurality of telephone lines and include codecs converting analog voice signals received over an activated telephone line to corresponding digital voice signals and converting digital voice signals generated by the voice service system to corresponding analog voice signals for communication over an activated line and the control system includes a digital switch coupled for communication with the line interface circuits and the store and selectively intercoupling digital representations of voice signals being communicated over different ones of the plurality of telephone lines with corresponding message baskets stored by the store.

14. The automated telephone voice service system according to claim 1 above wherein the control system responds to data signals appearing on the particular telephone line defining a CHANGE command by suspending a current operating mode thereof and enabling response to data signals appearing on the particular telephone line which define a new operating mode.

15. The automated telephone voice service system according to claim 14 above wherein the control system responds to a CHANGE MESSAGE BASKET command and a message basket identification received over the particular telephone line following receipt of a CHANGE command over the particular telephone line by disassociating the particular message basket from the particular telephone line and associating the particular telephone line with a second particular message basket indicated by the message basket identification.

16. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets, the store being coupled to store and retrieve representations of voice messages at each of the plurality of individually addressable message baskets therein;

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, with the telephone lines including at least on direct incall line, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a voice message received over the particular telephone line, and to forward a voice message representation stored in the particular message basket to at least one other of the individually addressable message baskets; and

the control system including means for detecting when the particular telephone line is a direct incall line and responding to such detection by association the particular telephone line with a predetermined particular message basket and precluding association of the particular telephone line with any other message basket, the control system being operable to enable only the message basket activity of recording a voice message received over the particular telephone line when the particular telephone line is a direct incall line.

17. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets, the store being coupled to store and retrieve representations of voice messages at each of the plurality of individually addressable message baskets therein; and

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, with the telephone lines including a direct incall line, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a voice message received over the particular telephone line, and to forward a voice message representation stored in the particular message basket to at least one other of the individually addressable message baskets, and

the control system including means for detecting when the particular telephone line is a direct recall line and responding to such detection by associating the particular telephone line with a predetermined particular message basket and precluding association of the particular telephone line with any other message basket, the control system being operable to enable an activity affecting the particular message basket only upon receipt over the particular telephone line of a predetermined personal identification code associated with the particular message basket when the particular telephone line is a direct recall line.

18. The automated telephone voice service system according to claim 17 above, wherein the predetermined personal identification code includes a first portion which cannot be changed in response to data signals received over the particular telephone line and a second portion which can be changed in response to data signals received over the particular telephone line.

19. The automated telephone voice service system according to claim 1 wherein the predetermined plurality of telephone lines includes a general access line with the control system including means for detecting that the particular telephone line is a general access line and responding to such detection by associating the particular telephone line with a particular message basket indicated by a message basket indication received over the particular telephone line as a data signal.

20. The automated telephone voice service system according to claim 19 wherein the message basket indication is alternatively a message basket code or a personal identification code having a predetermined association with the particular message basket and wherein the control system responds to the message basket code by enabling a voice message recording with respect to the particular message basket or responds to the personal identification code by enabling account ownership activities with respect to the particular message basket and further responds to commands received as data signals over the particular telephone line by executing any activity commanded thereby.

21. The automated telephone voice service system according to claim 20 wherein the account ownership activities include voice message retrieval and voice message sending.

22. The automated telephone voice service system according to claim 1 wherein the control system responds to a LISTEN command received over the particular telephone line by retrieving from the particular message basket and communicating over the particular telephone line any voice message whose representation is contained within the message basket and to a LISTEN command followed by a TALK command received over the particular telephone line by forwarding to any message basket from which a voice message representation retrieved in response to the LISTEN command has been forwarded a representation of a voice message received over the particular telephone line following the TALK command.

23. The automated telephone voice service system according to claim 1 wherein commands defined by the data signals received over the particular telephone line include an ATTENTION command and wherein the control system responds to an ATTENTION command received over the particular telephone line by entering a pause mode and enabling a response to additional commands received over the particular telephone line for a predetermined period of time following receipt of each ATTENTION command.

24. The automated telephone voice service system according to claim 1 wherein the store stores a client voice greeting that is uniquely associated with the particular message basket and wherein the control system responds to at least one type of access to the message basket by retrieving from the store and communicating over the particular telephone line the client voice greeting.

25. The automated telephone voice service system according to claim 24 wherein the data signals include a CHANGE GREETING command and wherein the control system responds to a CHANGE GREETING command received over the particular telephone line by storing in the store in place of any previously stored client voice greeting a new client voice greeting subsequently received over the particular telephone line.

26. The automated telephone voice service system according to claim 1 wherein the control system responds to data signals received over a particular telephone line commanding modification of a voice message which has previously been received by forwarding a representation of the received voice message to another message basket or attempting to forward the received voice message to another telephone line until the voice message has actually been communicated over a telephone line.

27. The automated telephone voice service system according to claim 1 above, wherein the control system forwards a representation of a voice message to at least one other message basket by retaining in the particular message basket a single copy of the voice message representation and storing in at least one of said other message baskets a pointer identifying the particular message basket and the voice message representation therein which is to be forwarded.

28. The automated telephone voice service system according to claim 2 wherein the store stores a plurality of different voice prompting messages which provide detailed explanations of voice service system usage for different status conditions of the voice service system and wherein the control system responds to data signals defining a PROMPT command by determining a current status condition of the voice service system and retrieving from the store and communicating over the particular telephone line a voice prompting message that is appropriate for the current status condition of the voice service system.

29. An automated telephone voice service system comprising:

a store having defined therein a plurality of individually addressable message baskets, the store being coupled to store and retrieve representations of voice messages at each of the plurality of individually addressable message baskets therein;

a control system providing a selective coupling between the store and each of a predetermined plurality of telephone lines of a telephone network, the control system being responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a representation of a voice message received over the particular telephone line, and to forward a voice message representation stored in the particular message basket to at least one other of the individual addressable message baskets;

an operator console coupled for communication with the control system; and

wherein the control system stores in the store each data signal received over the particular telephone line to provide a stored data signal audit trail and responds to data signals defining an OPERATOR ASSISTANCE command by providing to the particular telephone line a voice connection to the operator console and retrieving from the store and communicating to the console for display thereby the stored data signal audit trail.

30. The automated telephone voice service system according to claim 1 wherein the control system is further responsive to data signals received over the particular one of the telephone lines to forward a voice message whose representation is stored in the particular message basket to at least one telephone line different from the particular line.

31. An automated telephone voice service system comprising:

a store coupled to store and retrieve representations of voice messages at each of a plurality of individually addressable message baskets therein; and

a control system providing a selective coupling between the store and each of a given plurality of telephone lines of a telephone network, with a particular message basket being coupled to a particular telephone line in response to a set of message basket control signals received over the particular telephone line for storing in the particular message basket representations of a voice message received over the particular telephone line, responsive to a second set of message basket control signals for retrieving from the particular message basket and communicating over the particular telephone line a voice message whose representation has been previously stored in the particular message basket, responsive to a third set of message basket control signals which include an address of a forwarding message basket for forwarding a representation of a voice message that has been previ-

ously stored in the particular message basket to a forwarding message basket, and responsive to a fourth set of message basket control signals for forwarding a voice message whose representation has been previously stored in the particular message basket to a telephone line selected by the fourth set of message basket control signals.

32. The automated voice service system according to claim 31 above further comprising a voice prompting system coupled to communicate a voice message prompt explaining how to use the automated telephone voice service system upon activation of selected ones of the given plurality of telephone lines.

33. The method of providing a telephone voice service system response to an incoming telephone call from a caller on a telephone line comprising the steps of:

communicating over the telephone line a prerecorded voice message prompting the caller to enter alternatively a message basket code or a personal identification code;

determining the type of code entered by the caller;

if a message basket code is entered, prompting the caller to communicate a voice message whose representation is forwarded to a message basket identified by the code and storing in a message basket portion of a store indicated by the message basket code a representation of any voice message communicated by the caller;

if a personal identification code is entered, enabling account ownership functions for an account associated with the personal identification code including retrieval of messages from a message basket associated with the account and forwarding of message representations from the associated message basket to another message basket identified by signals communicated over the telephone line in accordance with a predetermined code.

34. In a voice message service system having means for storing a plurality of telephone voice messages and making such messages available to a plurality of calling parties on a real time basis, the combination comprising:

means providing a plurality of telephone lines for carrying voice data and command data;

means for processing data including means for storing digital representations of a plurality of voice messages with each message being addressably retrievable in response to an account identifier code corresponding thereto, including means responsive to a first command indicated by said command data for selectively retrieving a voice message identified by the command data and communicating the retrieved voice message over a telephone line from which the command data is received, including means responsive to a second command indicated by said command data for selectively retrieving a voice message identified by the command data and communicating the retrieved message over a telephone line indicated by the command data, which telephone line is not the telephone line from which the command data is received, including means for providing routine prestored prompting messages to a calling party, means for storing numeric data and command sequences in response to calls received over the telephone lines, means for recognizing invalid and inappropriate command sequences, means for initiating an operator assisted mode on the occurrence of an invalid or inappropriate command sequence, and means for generating data describing the status of messages pertaining to a given telephone number; and

at least one operator station responsive to the message status data from the data processing means and the initiation of the operator assisted mode for providing informed intervention in response to an occurrence of an invalid or inappropriate command sequence.

35. A system for storing and forwarding telephone calls comprising:

first interconnect means for subscribers, the first interconnect means being accessible in response to a number in a first group of numbers, each of which identifies a subscriber;

second interconnect means for non-subscribers, the second interconnect means being accessible in response to a number in a second group of numbers, each of which identifies a subscriber;

a central station including inbasket means for storing representations of voice messages intended for subscribers and outbasket means for storing representations of subscriber originated voice messages intended for conversion and transmission from the outbasket means to a telephone;

and means at the central station responsive to the first and second interconnect means and coupled to control the inbasket means and outbasket means to permit subscribers to access the inbasket means and control the outbasket means through the first interconnect means while permitting non-subscribers access only to the inbasket means through the second interconnect means for the purpose of recording a voice message for later retrieval by a particular subscriber identified by a number in the second group which is used to access the second interconnect means.

36. A voice message service system comprising:

data processing means for monitoring the status of subscriber messages and subscriber dialing operations, said date processing means including means for providing routine prestored prompting messages to a party calling the system, means for storing representations of voice messages, numeric data and command sequences in response to calls, means for recognizing invalid and inappropriate command sequences, means for initiating an operator assisted mode on the occurrence of an invalid or inappropriate command sequence and means for generating data describing the status of messages pertaining to a given telephone number; and

at least one operator station including display means responsive to the message status data from the data processing means and the initiation of the operator assisted mode for providing informed intervention on the occurrence of an invalid or inappropriate command sequence.

37. The voice message service system as set forth in claim 36 above, wherein the data processing means includes means responsive to a direct operator request command from a subscriber for providing message status data to the operator station and switching the call to the operator station.

38. The voice message service system as set forth in claim 37 above, wherein the data processing means comprises means for communicating to the display means for displaying thereon, data indicating a command sequence entered by a subscriber.

39. A telephone subscriber service system comprising:

a telephone line concentrator connected at a central office of a telephone network to a line of each service system subscriber and operable upon the occurrence of a predetermined condition on a subscriber line to connect the subscriber line to a trunk line;

at least one trunk line connected between the concentrator and a trunk interface circuit;

a trunk interface circuit connected between the at least one trunk line and a computer and communications system, the trunk interface circuit being operable to couple to the computer and communications system digital data representative of analog signals appearing on the trunk line and to couple to the trunk line analog signals representative of digital data received from the computer and communications system;

a line interface circuit coupled between a telephone company telephone line and the computer and communications system, the line interface circuit being operable to couple to the computer and communications system digital data representative of analog signals appearing on the telephone line and to couple to the telephone line analog signals representative of digital data received from the computer and communications system;

the computer and communications system coupled for bidirectional communication with the trunk interface circuit and line interface circuit, the computer and communications system having established therein means for storing providing an inbasket and an outbasket associated with and controlled by each service system subscriber, with each inbasket providing storage for representations of messages received from a calling party for the associated subscriber and each outbasket proving storage for representations of messages received from the associated subscriber for delivery to another party.

40. The telephone subscriber service system according to claim 39 above, wherein the computer and communications system includes means for controlling coupled to control the communication of voice messages between the storing means and a telephone network user, the controlling means including means for detecting the occurrence of tone signals indicative of the actuation of corresponding keyboard keys and means responsive to detection of a tone signal corresponding to actuation of a predetermined keyboard key for enabling the controlling means to control the operation of the subscriber service system in response to the detection of at least one additional tone signal corresponding to actuation of a keyboard key.

41. The telephone subscriber service system according to claim 39 above, wherein the computer and communcations system includes means for controlling coupled to control the communication of voice messages between the storing means and a telephone within the telephone network, the controlling means including means for detecting the occurrence of tone signals indicative of the actuation of corresponding keyboard keys and means responsive to a predetermined combination of a plurality of different ones of said tone signals including a combination of tone signals identifying a user of the telephone as a particular system subscriber and including at least one tone signal indicating a talk command, for storing in an outbasket associated with said particular system subscriber a representation of a voice message received from said telephone.

42. A telephone subscriber service system comprising:

an interface circuit coupled to provide bidirectional communication between a telephone network including tone signal producing keyboard telephones and a digital data processing system, with information being communicated between the interface circuit and the data processing system in digital form and between the interface circuit and the telephone network in a form compatible with the operation of the telephone network;

the digital data processing system coupled to the interface circuit and including:

means for providing communication of voice messages with telephone system users,

means for storing voice messages in digital form for each different subscriber to the subscriber service system, and

means for controlling coupled to control the communication of voice messages between the storing means and a telephone network user, the controlling means including means for detecting the occurrence of tone signals indicative of the actuation of corresponding keyboard keys and means responsive to detection of a tone signal corresponding to actuation of a predetermined keyboard key for enabling the controlling means to control the operation of the subscriber service system in response to the detection of at least one additional tone signal corresponding to actuation of a keyboard key, means for limiting all functions by a non-account caller to the storage of a voice message and editing of a voice message being stored during the course of a single call, and means responsive to receipt of a data code over a telephone line of the telephone network identifying a caller as an account owner for enabling access by the caller to and execution of account ownership functions including (1) retrieval of account messages, (2) digital recording of account messages for delivery to one or more other accounts and (3) automatic storage and delivery of a voice reply to a caller originating a message in response to an ATTENTION, TALK command sequence during operation in a message retrieval mode in which a message from another account is retrieved.

43. An automated telephone voice service system comprising a data processing system coupled to receive, store and retrieve representations of voice messages received over a telephone line and to respond to tone commands received over the telephone line, the data processing system being operative to limit call functions by a non-account owner caller to the storage of representations of a voice message and editing of the voice message during the course of a single call and including means operative in response to a data code received over the telephone line identifying a caller as an account owner for executing account ownership functions including retrieval of account messages, recording of account messages for delivery to one or more other accounts, and in a message retrieval mode in which a message from another account owner is retrieved, automatic storage and delivery of a voice message reply in response to an ATTENTION, TALK command sequence.

44. The automated telephone voice service system according to claim 43 above wherein the data processing system includes means for executing account ownership functions which include in a message retrieval mode, the saving of a current message for later recall in response to an ATTENTION, SAVE command sequence.

45. The automated telephone voice service system according to claim 43 above, wherein the data processing system includes means for granting a caller having access to a given account access to another account in response to a command sequence ATTENTION, CHANGE, CHANGE.

46. The automated telephone voice service system according to claim 43 above, wherein the data processing system includes means for processing an account address either in the form of an account number identifying a third party account, or a code identifying a preestablished address list having at least the third party account identified thereon.

47. The automated telephone voice service system according to claim 43 above, wherein the data processing system includes means for processing a received data code which includes a first code identifying the owned account and a second, personal identification, code preceded by an ATTENTION command identifying the caller as the owner of the owned account.

48. The automated telephone voice service system according to claim 43 above, wherein the data processing system includes means operable in any mode in any mode for enabling a caller to leave a voice message with another account in response to a command sequence including ATTENTION, CHANGE, TALK, ACCOUNT NUMBER of the account which is to receive the message.

49. An automated telephone voice service system comprising:

a store coupled to store and retrieve representations of voice messages at each of a plurality of individually addressable message baskets therein; and

a control system providing selective coupling between the store and each of a plurality of telephone lines of a telephone network with at least one of the lines being a general access line over which a plurality of different message baskets may be accessed for either message storing or account ownership functions, with a message storing function being enabled in response to entry of a code identifying one of the plurality of message baskets and account ownership functions being enabled in respone to entry of a code identifying one of the plurality of message baskets and a personal identification code identifying the owner of the one message basket.

50. The automated telephone voice service system according to claim 49 above, wherein the control system is operative to respond to a command series ATTENTION, CHANGE, CHANGE by enabling receipt of a different message basket identification code identifying a message basket different from a currently accessed message basket and granting access to the different message basket in response to the different code.

51. The automated telephone voice service system according to claim 49 above, wherein the control system includes means for receiving and responding to account ownership administrative commands after said administrative commands are enabled by a command sequence including ATTENTION, CHANGE, ADMINISTRATION.

52. The automated telephone voice service system according to claim 51 above wherein the control system includes means for distinguishing a given code for at least one administrative command which may be entered after administrative commands are enabled from the same given code for a nonadministrative command which may be entered when administrative commands are not enabled.

53. An automated telephone voice service system comprising:

a store coupled to store and retrieve representations of voice messages at each of a plurality of individually addressable voice message baskets therein, the message baskets being arranged in at least first and second groups with each message basket address having first and second fields, the message baskets of the first group having a first group first field address and mutually exclusive second field addresses, and the message baskets of the second group having a second group first field address and mutually exclusive second field addresses; and

a control system providing a selective coupling between the store and each of a plurality of telephone lines which provide a signal at the beginning of each incoming call which identifies a telephone number of a telephone from which the call is being placed, the control system being operative to associate a first group of telephone numbers with the first group of message baskets and upon receiving a second field message basket address from a telephone having a first group telephone number to couple the call to the addressed message basket within the first group of message baskets and upon receiving a second field message basket address from a telephone having a second group telephone number to couple the call to the addressed message basket within the second group of message baskets, the control system being further operative to store and retrieve voice messages communicated between a coupled message basket and a calling telephone.

54. The automated telephone voice service system according to claim 53 above, wherein the control system is further responsive to a message basket address received from a telephone line, which message basket address contains both a first field address and a second field address, by coupling the telephone line to a message basket indicated thereby.

55. A telephone voice message service system comprising:

an information processing system operative to receive, store and retrieve digital samples representing voice messages and to command the selective interconnection of channels carrying sequences of digital samples which each represent a voice message; and

a real time subsystem coupled for communication with a plurality of bidirectional voice communication channels carrying sequences of digital samples with each sequence representing a voice messages, the real time subsystem being responsive to information processing system commands to selectively couple data samples received from any channel or from the information processing system to any channel or from any channel to the information processing system to provide commanded interconnection of the voice communication channels.

56. The telephone voice message service system according to claim 55 above, wherein the real time subsystem includes a time division multiplexed real time bus and selectively interconnects the bidirectional voice communication channels and the information processing system by placing incoming digital samples on the bus at commanded periodic sample data time intervals for each channel and taking outgoing digital samples off the bus at commanded periodic sample data time intervals for each channel.

57. The telephone voice message service system according to claim 56 above, wherein the real time subsystem further includes at least one real time processor coupled between and communicating digital samples between the real time bus and the information processing system, the real time processor being operative to process the digital samples communicated thereby to provide at least one real time processing function including silence detection and compression.

58. The method of telephone voice message communication comprising the steps of:

answering a telephone line;

receiving over the answered telephone line an identification code which identifies the caller as a subscriber having a subscriber message basket for storing data which includes representations of voice messages, the message basket having an inbasket portion and an outbasket portion;

receiving over the answered telephone line a first signal indicating at least one command including a talk command;

receiving over the answered telephone line and storing in the outbasket portion of the subscriber message basket in response to the talk command a representation of a voice message generated by the caller;

receiving over the answered telephone line a second signal including information identifying at least one designated recipient of the voice message; and

for each designated recipient:

calling the designated recipient by dialing a designated recipient telephone line corresponding to the designated recipient,

when the designated recipient telephone line is answered, communicating over the designated recipient telephone line a voice message delivery greeting including an explanation that a recorded voice message is being delivered,

retrieving from the outbasket portion of the subscriber message basket and communicating over the recipient telephone line the voice message, and

terminating the call to the designated recipient.

59. The method of telephone voice message communication according to claim 58 above, further comprising between the step of retrieving and communicating and the step of terminating, the step of receiving a reply message to the communicated voice message.

60. The method of telephone voice message communication according to claim 59 above, wherein the step of receiving a reply message includes the steps of:

communicating over the designated recipient telephone line a reply invitation voice message;

storing in the voice message store a representation of a voice reply message received over the designated recipient telephone line; and

communicating over the designated recipient telephone lines a voice delivery closure message.

61. The method of telephone voice message communication according to claim 60 above, wherein the step of receiving and storing includes the step of receiving editing commands communicated over the designated recipient telephone line and editing the voice reply message in accordance with received editing commands.

62. The method of telephone voice message communication according to claim 61 above, wherein the reply invitation voice message includes an indication that editing commands may be used in creating the reply voice message.

63. The method of telephone voice message communication according to claim 58 above, wherein the voice message delivery greeting is a voice message having a representation thereof previously stored and is automatically retrieved and automatically communicated over the designated recipient telephone line without human intervention.

64. The method of telephone voice message communication according to claim 63 above, wherein the second signal includes information identifying at least one predetermined list of designated recipients and a predetermined set of delivery instructions.

65. The method of telephone voice message communication according to claim 64 above, wherein the predetermined set of delivery instructions includes information indicating at least one period of time during which a designated recipient telephone line is to be dialed and wherein the step of calling includes the steps of automatically dialing without human intervention the designated recipient telephone line during a period of time indicated by the predetermined set of delivery instructions.

66. The method of telephone voice message communication according to claim 65 above, wherein the predetermined set of delivery instructions includes information indicating a retry time interval and a maximum number of retry attempts when a dialed call is not completed and further comprising a step of automatically redialing the designated recipient telephone line in accordance with the set of delivery instructions when a dialed call is not completed.

67. The method of telephone voice message communication according to claim 58 wherein the at least one designated recipient includes a subscriber having an identification code and an associated message basket having an inbasket portion and an outbasket portion and further comprising the step of storing in the inbasket portion of the message basket of the designated recipient subscriber information identifying the voice message and the location at which a representation of the voice message is stored.

68. The method of providing a telephone voice service comprising the steps of:

answering incoming calls and accepting and responding to caller originated commands for voice message operation including commands to record or retrieve voice messages or in the absence of receiving a caller originated command within a predetermined time period,

generating a voice message salutation which invites the caller to leave a voice message following a tone signal;

pausing after generating the saluation;

generating a tone after pausing; and

recording a representation of any voice message communicated after the tone is generated.

69. The method of providing a telephone voice service according to claim 68 above wherein a caller originated command is recognized as a conmand only when preceded by a predetermined ATTENTION code signal.

70. The method of providing a telephone voice service according to claim 68 above, further comprising the step of interrupting any current voice service activity related to a given call upon receipt of an ATTENTION code signal from the caller and awaiting receipt of a further command signal from the caller.

71. The method of providing a telephone voice messaging service through a telephone voice service system comprising the steps of:

providing for each client a message basket for storing client related information including representations of voice messages that may be edited by the client while being stored in a client message basket, each message basket having an inbasket portion and an outbasket portion;

receiving and recording in a sending client outbasket portion of the client message basket, a single copy of a representation of a voice message;

receiving and storing an indication of a plurality of destinations to which the voice message is to be delivered;

delivering the voice message to each indicated destination by retrieving the single copy of the representation of the voice message from the outbasket portion of the client message basket for each delivery and communicating the voice message to one of the plurality of indicated destinations.

72. The method of providing a telephone voice messaging service according to claim 71 above, wherein the delivering step includes, when the voice message is to be delivered to a system inbasket of a recipient person who is a system client, the steps of placing in the inbasket of the recipient person a cross reference address to the storage location of the single voice message representation copy in the outbasket, and accessing the single voice message representation copy using the address cross reference to retrieve the voice message representation for delivery of the voice message to the system client recipient person.

73. The method of providing a telephone voice messaging service according to claim 71 above further comprising the steps of maintaining for each recipient person an indication of whether or not the voice message has been delivered to the recipient person and enabling a sending client to edit the voice message at any time until the voice message has been delivered to every recipient person.

74. The method of providing a telephone voice messaging service through a telephone voice service system comprising the steps of:

providing for each client a message basket for storing client related messages, each message basket including an addressable outbasket;

receiving and recording in a sending client outbasket a single copy of a representation of a voice message;

receiving and storing an indication of a plurality of destinations to which the voice message is to be delivered;

delivering the voice message to each indicated destination by retrieving the single copy of the representation of the voice message from the outbasket of the client message basket for each delivery and communicating the voice message to one of the plurality of indicated destinations; and

temporarily spacing the delivery of the voice message to each different recipient person by a predetermined time interval to avoid congestion of the telephone voice service system.

* * * * *