digital

VAXELN Technical Overview

VAXELN
VAXELN
VAXELN
VAXELN
VAXELN
VAXELN
VAXELN
VAXELN

Contents

What is VAXELN?

VAXELN is a software product for the development of dedicated, realtime applications for the VAX family of super minicomputers.

For the purpose of this discussion, a dedicated application is one in which the computer is used to solve a specific problem or set of related problems. The term spans a wide range of applications, including workstations designed for a particular profession, automated industrial machinery and robots, process control and simulation.

A realtime application is one in which the system's response to external events is critical. VAXELN was designed with realtime applications in mind, and imparts as little "overhead" as possible, thereby assuring the most speed and responsiveness from the VAX processor.

Since they are optimized for speed and efficiency, VAXELN programs are perfect for areas where general-purpose operating systems are not suited. VAXELN systems include only those services that are needed to support the functions required by the application. In other words, VAXELN systems are only as complex as they need to be to do their work. Such systems are called "statically defined".

The VAXELN Toolkit is a layered product that runs on any VAX processor under the VMS operating system (V4.0 or later) or MicroVMS (V4.0 or later). The Toolkit consists of a highly optimized, efficient kernel executive, a system image builder, a remote symbolic debugger, an extended Pascal compiler (EPascal) and runtime libraries that support EPascal, VAX C, and VAX FORTRAN 77. Ada/ELN and its runtime libraries are supported as a separate option. During program development, standard VMS utilities such as the VMS editors, linker and library facilities are used to provide the simplest, and yet most comprehensive development environment available in the industry.

The development system, also known as the host, runs the VAXELN Toolkit under the VMS general-purpose multi-user operating system. The application system, known as the target, is executed under the control of the VAXELN kernel, which, along with other Digital-supplied components, deals solely with the application at hand; i.e. a VAXELN system is dedicated to the application and should not be considered a general-purpose operating system. Completed VAXELN applications can be loaded from FILES-11 format disk media, or, if the optional DECnet-VAX license and Ethernet hardware are present, downloaded from the host to the target. Additionally, any VAXELN application may be "blasted" into read-only memory (ROM), through use of an optional utility, for later installation and execution on the target system.

One of the more significant optimizations in VAXELN is in the development of device drivers. VAXELN provides the ability to write drivers in a high-level language (EPascal, C or Ada®) and supplies templates for the commonly used drivers as part of the Toolkit. This means development time for your applications is drastically reduced. Machine language programming is not required.

Traditionally, the application developer had to be an expert in several fields: design of realtime systems, hardware architecture, programming in machine language and operating systems internals. The design of VAXELN eliminates, as much as possible, those areas of the development process not directly associated with the specific target application, and allows the developer to concentrate on only the application. VAXELN provides the following features and capabilities in pursuit of this goal.

- High-level programming languages
- Concurrent execution of programs
- Transparent Ethernet support
- Target system debugging

High-Level Programming Languages

VAXELN systems can be developed entirely in high-level languages. VAXELN supports an extended version of ANSI/ISO Pascal (EPascal), VAX C or Ada/ELN (a fully compliant version of ANSI-MIL-STD-1815-A-1983 Ada). You may also include modules written in VAX FORTRAN 77 in your application. EPascal, C and Ada/ELN based applications have the ability to handle all hardware devices, exceptions, timeouts and even power failures. The need to spend weeks or even months developing and debugging drivers in machine language is eliminated.

Concurrent Execution of Programs

VAXELN provides concurrent execution, i.e., a program made up of several concurrently executing parts of EPascal, C, and Ada/ELN Programs. A full definition of concurrent programming is beyond the scope of this discussion. However, its most basic principle is that component parts of a program are allowed to

Ada is a registered trademark of the U.S. Government.

execute simultaneously (multitasking) and programs within an application system are also allowed to execute in parallel (multiprogramming).

Even in cases where the programs and/or program components do not actually share the same processor (and so do actually execute in parallel), concurrent programming has numerous advantages in system design, including improved performance, compared with simpler models in which every program runs to completion before any other can run.

Transparent Ethernet Support

Local area network support based on Ethernet is designed into VAXELN's basic architecture. Data transmission facilities are provided to make it easy to distribute an application's component programs among several participating network nodes. Changing the node location of a program does not require rewriting the program. Programs initially written to execute on the same processor may be distributed among nodes on the network (such as when the application expands and more processing power is required).

Debugging the Target System

Once the application software has been written, it may be tested and debugged in one of two ways. The VAXELN Toolkit provides a remote symbolic debugger which allows the programmer to debug the target application from the host computer via an optional Ethernet connection. If the Ethernet is not available, the application may be debugged directly on the target hardware via the local, nonsymbolic debugger included in the Toolkit.

What is a VAXELN System?

A VAXELN system is a set of programs executing on VAX hardware, including code and data provided by both Digital and the application developer. Diagramatically, a typical VAXELN system might be represented as shown in Figure 1.

The hardware includes one or more VAX processors in a host-target relationship, plus optional peripheral devices including disks, terminals, communication hardware, and special interfaces as defined by the specific application. The target VAX can exist as a stand-alone system or included in a local area network as the logistics of the application require.

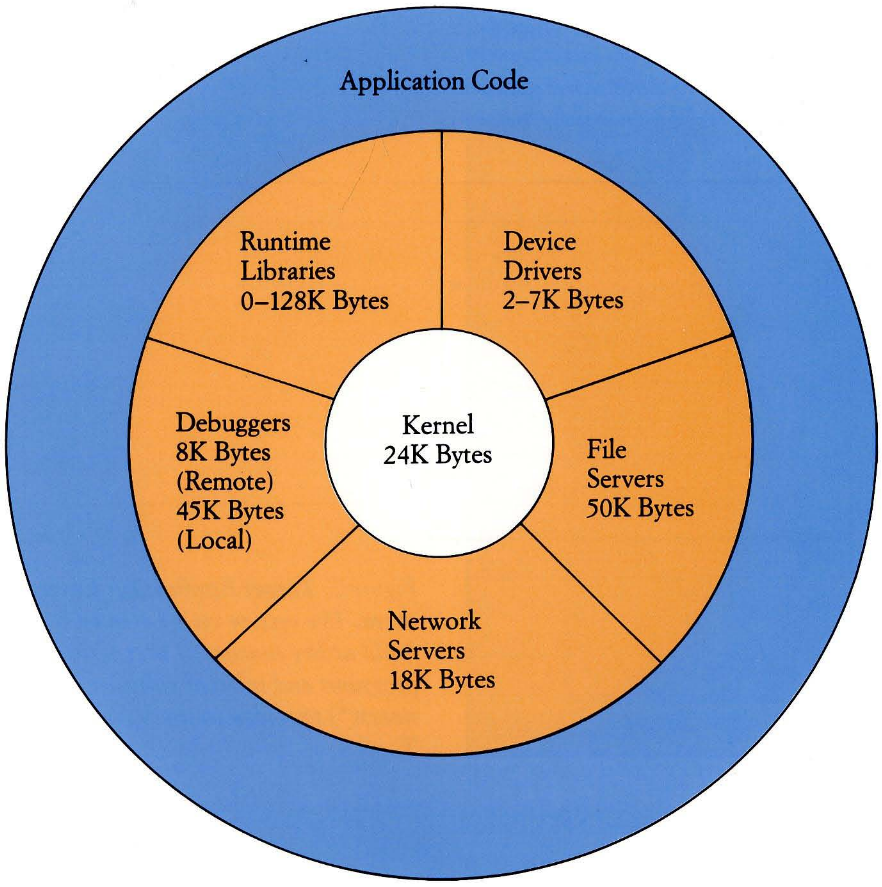
Application Code
Runtime Libraries 0–128K Bytes
Device Drivers 2–7K Bytes
Kernel 24K Bytes
Debuggers 8K Bytes (Remote) 45K Bytes (Local)
File Servers 50K Bytes
Network Servers 18K Bytes

Figure 1. ***VAXELN System Elements***

Essentially, programs executing in a VAXELN system may be classified as one of two types:

- User Programs—written in a high-level language, including, for example, data acquisition and reduction programs, process control supervisors, user-written device drivers, etc.
- Digital Supplied Programs—including the kernel executive, the network and file servers (if required), device drivers for standard Digital peripherals, and runtime libraries.

When combined, using the Toolkit's system build utility, the user- and Digital-supplied programs comprise an application system. See Figure 2.

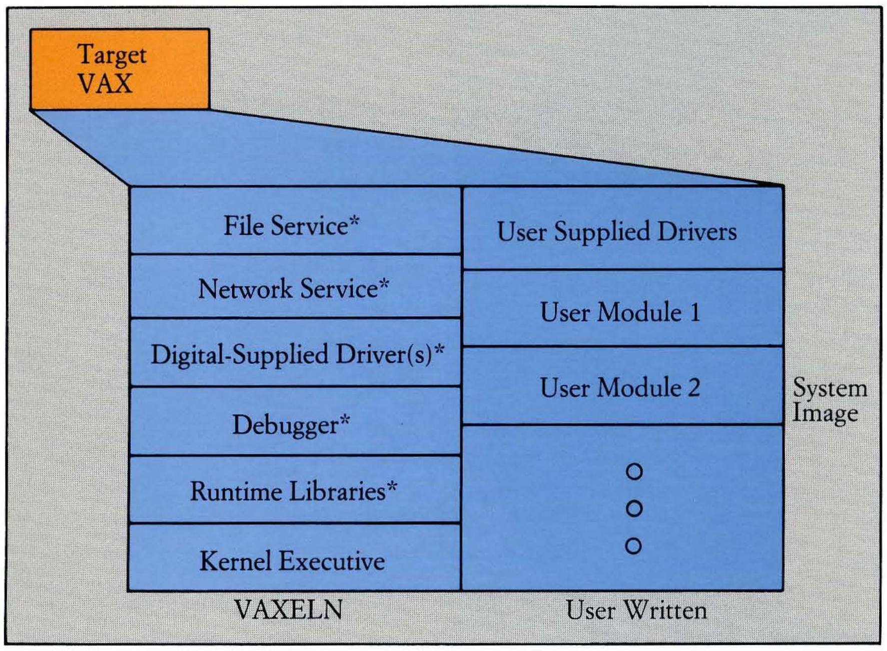

Figure 2. ***Target Application Environment.*** *The system image created by the build utility consists of user-written programs and selected software components(*) provided in the VAXELN Toolkit.*

How do I Use VAXELN?

The VAXELN Toolkit has been described as "sophisticated tools for the computer craftsman". Indeed, VAXELN offers both the "friendly" yet powerful development tools provided by VAX/VMS and the elegant simplicity and efficiency of the VAXELN runtime environment.

The Development Environment

The VAXELN development environment requires the following hardware and software components in order to produce an executable VAXELN application:

- Hardware: Any member of the VAX family of computers, supporting VMS/MicroVMS V4.0 or later.

  Optional Ethernet interfaces and the appropriate interconnecting hardware are needed to support downline loading and symbolic debugging.

  Optional peripheral devices from Digital, third party, or users may be needed for some applications.

- Software: VAX/VMS or MicroVMS V4.0 or subsequent releases.

  Optionally, DECnet-VAX V4.0 or subsequent release (for remote debugging and/or downline loading of application software).

  VMS compilers to support development of applications in the following languages:

  VAX C Compiler V2.1 (or subsequent release).

  VAX FORTRAN 77 V4.4 (or subsequent release).

  Ada/ELN Compiler and Runtime Libraries V1.0 (or subsequent release).

  Rdb/ELN V1.2 (or subsequent release for using Rdb/VMS compatible relational database manager in VAXELN applications).

Also available are the VAX Language Sensitive Editor (LSE) V1.2 or subsequent release (for creating programs in an editing environment tailored for software development in high-level languages) and DECprom V1.1 or subsequent release (for programming EPROMs with VAXELN application images).

The VAXELN development cycle has three distinct parts: design and coding, debugging, and system integration. VAX/VMS and the VAXELN Toolkit provide all the necessary tools to satisfy the needs of the most sophisticated developer.

Design and Coding ☐ To facilitate the design of a complex, realtime application, VAXELN employs features rarely seen in other realtime software products. Oversimplified, a VAXELN application may be defined totally within the context of a structured, modular, high-level language such as EPascal, C or Ada/ELN. Language constructs that implement VAXELN architectural features are either predefined in the compilers or provided as callable routines in run-time libraries.

Once compiled, the VAXELN application programs are processed by the standard VAX/VMS linker, merged with the selected Digital-supplied components, (such as the kernel executive, the network and/or file service and appropriate device/interface drivers) to comprise a VAXELN System Image. The image is a direct address-for-address image of the final target system.

Debugging ☐ Once the program has been successfully built, it is then necessary to transfer it to a target environment, in which it may be debugged or checked out. VAXELN supplies two debugging environments which may be used depending on the hardware or logistical considerations of the application.

As shown in Figure 3, the remote, symbolic debugger provided with the Toolkit allows the developer to downline load (given that the appropriate Ethernet hardware and DECnet-VAX software are in place) and run the application software on the target VAX from the host VAX. Breakpoints can be set, variable contents can be examined and modified, and code can even be altered via source listings (instead of compiler generated assembler listings).

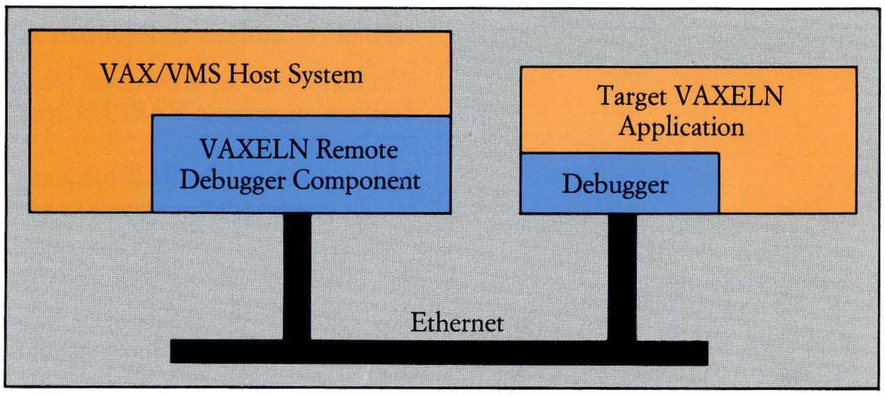

Figure 3. ***Remote Debugging Environment.*** *A target VAXELN application can be debugged from the host development system using the remote, symbolic debugger supplied with the Toolkit.*

Whereas most other debuggers will cause the entire system to halt if a breakpoint is encountered in any given module, VAXELN allows all but the affected module to continue running as scheduled while the suspect module is investigated. (Note that if there is a synchronization dependency between the affected module and any other active module in the system the results may be unpredictable.)

If it is not feasible to debug the application with the remote debugger due to physical distance between the host and target or lack of Ethernet capability, the Toolkit also provides a local, non-symbolic debugger which provides all of the traditional debugging functions such as examine/deposit memory/registers, set breakpoints, etc. from the local target console terminal. See Figure 4.

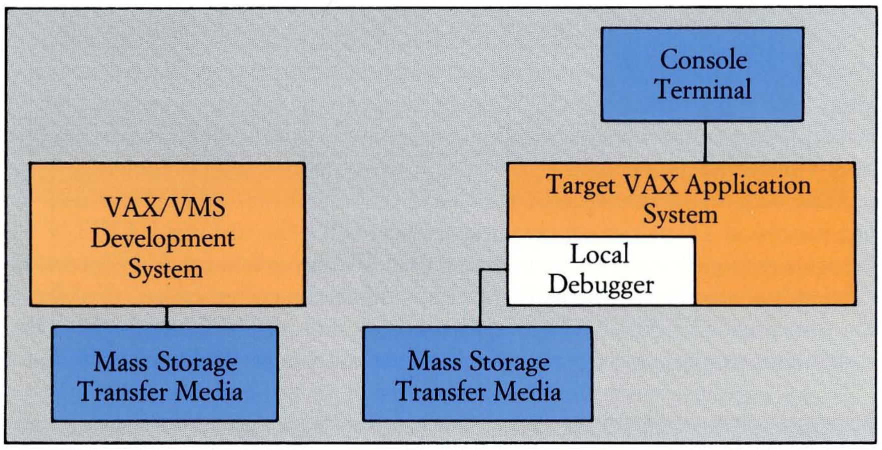
Console Terminal
VAX/VMS Development System
Target VAX Application System
Local Debugger
Mass Storage Transfer Media
Mass Storage Transfer Media

Figure 4. ***Local Debugging Environment.*** *When the application is not included in a local area network, it can be debugged with the local debugger.*

System Integration □ An individual program or part of an application may be debugged independently of many of the other components of the final system (for example, a driver for a specific device or a computation-intensive module). However, it is at some point in the development cycle when all the components must be brought together and exercised as a whole.

The Runtime Environment

The VAXELN runtime environment requires the following hardware and software components:

- Hardware: VAX 11/730, 11/750, VAX 8500, 8550, 8700, 8800, or MicroVAX II, KA620.

  Optional Ethernet hardware is required for downline loading of VAXELN system images.

  Optional Digital, third party or user-supplied peripheral devices may be required for some applications.
- Software: A properly developed and debugged VAXELN system image consisting of the VAXELN kernel executive plus device drivers, file and network services and user code as required by the application and hardware configuration.

VAXELN Programming Concepts

Development of a VAXELN application begins with a design based on the concept of concurrent execution of programs, also known as concurrency. Concurrency is a proven approach for such applications as multiple programs working together at the same time to solve a problem; production shop floor management, aircraft cockpit simulation or accommodating a number of professional workstations. This concept of concurrency has been built into VAXELN.

Programs in a VAXELN system are known as jobs. Multiple programs, or jobs, comprise a typical VAXELN-based application. Within a job exist functionally independent components known as processes. The ability of these processes to operate independently and in parallel (concurrently) is known as multitasking, and is managed by the VAXELN kernel executive. While only one process at a time can execute in any single CPU, the realtime VAXELN kernel is responsible for seeing to it that the CPU (as well as other resources) is efficiently shared by all processes in the system. As described earlier, the jobs that comprise a VAXELN system may reside on a single VAX processor, or be distributed among several processors in a local area network (Figures 5 and 6).

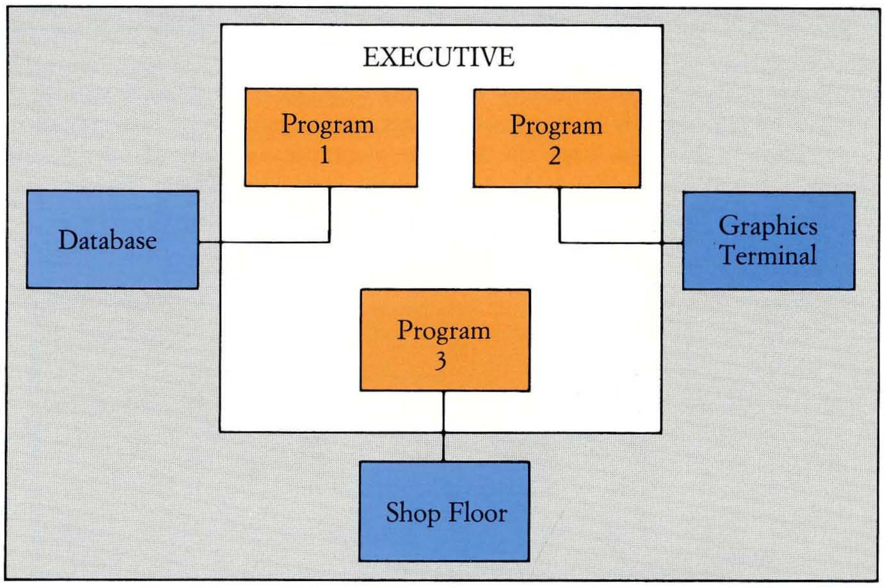

Figure 5. ***Single Processor Multiprogramming.*** *In multiprogramming, more than one part of an application can run on a single processor.*

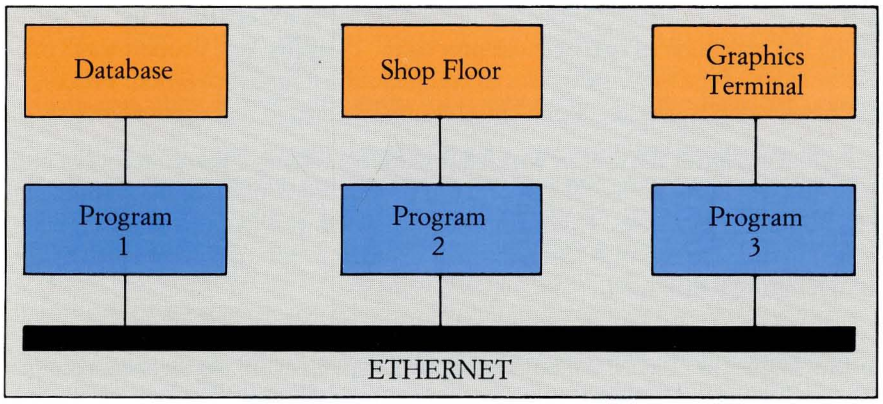

Figure 6. ***Distributed Processing Multiprogramming.*** *In distributed multiprocessing, more than one application can run on two or more processors connected together in a local area network.*

Multitasking □ Multitasking is the level of concurrency where each program in the application is divided into the necessary number of tasks. There is a separate process written for each task to be performed, each process concentrating on its own task. Processes may or may not execute concurrently, depending on the design of the program and the application at hand. While some processes are unable to execute (i.e. while they are waiting for some event or a resource to become available) other processes may execute and perform useful work for the application. See Figure 7.

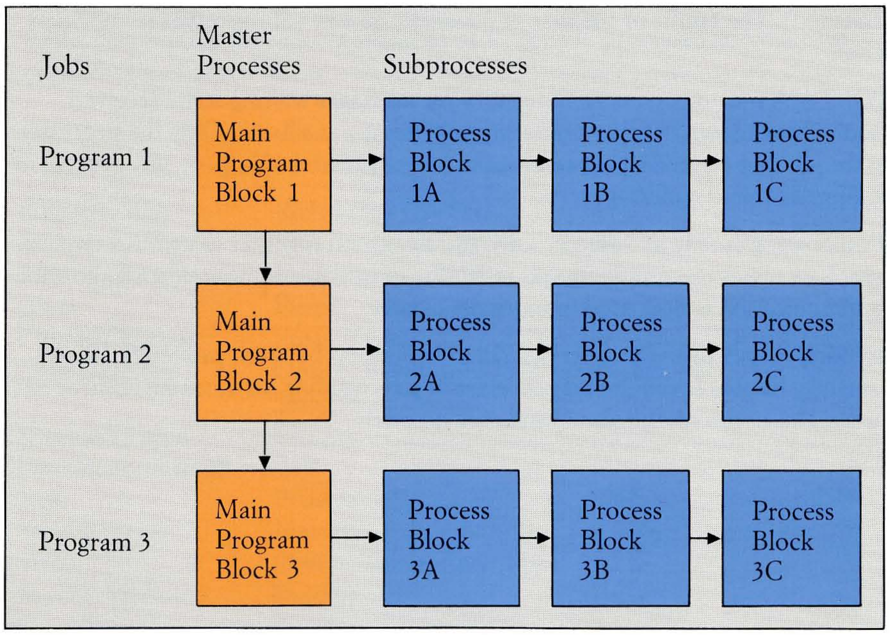

Figure 7. ***Master Process/Subprocess Relationship.*** *Jobs that execute application programs usually consist of a master process and one or more concurrently executing subprocesses.*

Process Synchronization □ Process synchronization is a mechanism for coordinating the execution of two or more processes. Typically, one process waits for one or more of the other processes to complete some finite operation. Process synchronization is needed in two cases: mutual exclusion and event response.

Mutual exclusion is the case in which a process is utilizing a shared resource and must have exclusive access to it in order to prevent corruption of the resource (a data buffer, for example). Event response is needed when a process must be activated in order to respond to a particular external or internal event (such as the completion of a disk read).

Objects □ Process synchronization is most commonly implemented by use of entities known as objects. An object may be a flag, an event occurrence, receipt of a message, etc.. VAXELN processes use objects in synchronizing their execution with the events of the realtime application. Table 1 is a list of objects available under VAXELN and a short description of each.

Table 1 VAXELN Objects

PROCESS—A process object represents an independent thread of execution of a code segment (that is the process described earlier). In VAXELN there is a master process (also known as the program) which controls one or more subprocesses. There can be any number of master processes executing the same subprocess (thus VAXELN code generated by the compilers is termed multithreaded). The family of the master processes and its subprocesses is known as a job.

PORT—A port represents a repository for messages waiting to be received (sometimes known in other operating systems as a mailbox). Only the processes in the job that created a port can receive a message from that port; any process in any job can send a message to it.

Two ports, possibly in different jobs, can be bound together to create a circuit, which increases the simplicity and reliability of interjob communication. Interjob communication is discussed in more detail later.

MESSAGE—A message describes data transmitted between processes. A message may be sent between two processes or jobs residing on the same node or between two nodes on the same local area network.

NAME—Ports may be assigned a name (e.g. Fred) which is known either locally (only to jobs physically executing on a specific system, or node) or universally (to all jobs executing on any node in the local area network comprising the system).

SEMAPHORE—The semaphore represents a synchronization gate used to meter process execution and synchronize access to shared resources. A semaphore can be of two types, binary and counting. A binary semaphore may assume only one of two values: 0 or 1. A counting semaphore may assume any value from 0 to 4 billion and is usually meant to keep track of the number of pending accesses to a particular resource.

EVENT—An event object represents the state of an event used for process synchronization access to shared data. A typical event might be the completion of a code segment or the completion of a disk access.

DEVICE—A device represents a device interrupt connected to an interrupt service routine or driver.

AREA—An area represents an amount of physical memory globally accessible by all jobs within the same node.

To use these objects in programming a VAXELN application, the kernel provides a number of operations or kernel procedures which, in turn, manipulate the objects and report back to the issuing program the status or result of the manipulation.

Table 2 represents the objects available to VAXELN processes and the operations that may act on them.

Table 2 VAXELN Operations and Objects

OPERATION	*RELATED OBJECT*
Accept_Circuit	Port
Allocate/Free	Memory
Clear	Event
Connect_Circuit	Port
Create/Delete	Device, Event, Job, Message, Name, Port, Process, Semaphore
Disable/Enable	Process
Disconnect_Circuit	Port
Exit	Process
Send/Receive	Message
Signal	Device, Event, Port, Time
Suspend/Resume	Process
Translate	Name
Wait_Any/Wait_All	Device, Event, Port, Process, Semaphore, Time

The VAXELN Kernel

The kernel is the layer of software that lies between the hardware and the application software. It is the kernel that provides direct manipulation of system objects as directed by application programs through the use of the operations described above. The kernel thus provides for the controlled sharing of system resources and synchronizes communication among the various programs in the system. The kernel also is responsible for maintaining all information about the system as a whole and each program component (known as the system or program's context).

A VAXELN System □ Any number of executable programs (jobs) can be combined with the kernel to form a system. When a system is built, there is the option to specify any number of programs that will be executed automatically when the system is initially loaded on the target VAX.

As mentioned previously, each job consists of a master process and zero or more subprocesses, which can execute independently in parallel, or concurrently. Processes are created dynamically by the master process or any of the subprocesses. Once created, a process remains active until it terminates normally or is terminated as a result of the delete operation.

Job termination normally occurs when the master process finishes executing its code segment(s). The termination of the master process (or any process for that matter) can also occur as a result of the signal or exit operations. Signalling a process raises a special exception for that process which is handled much as other hardware and software exceptions (such as divide by zero). The process may then take special measures if need be (e.g. close all open files) before exiting.

A process can also delete itself or any other process within the same job. Deleted processes cannot be restarted; in general, signal and exit provide a more controlled means of forcing a process to terminate.

Job termination (i.e. when the master process terminates) also means that all of its subprocesses and shared data are deleted from the system.

Process States □ Each process in a VAXELN system is always in one of the following process "states" (see Figure 8):

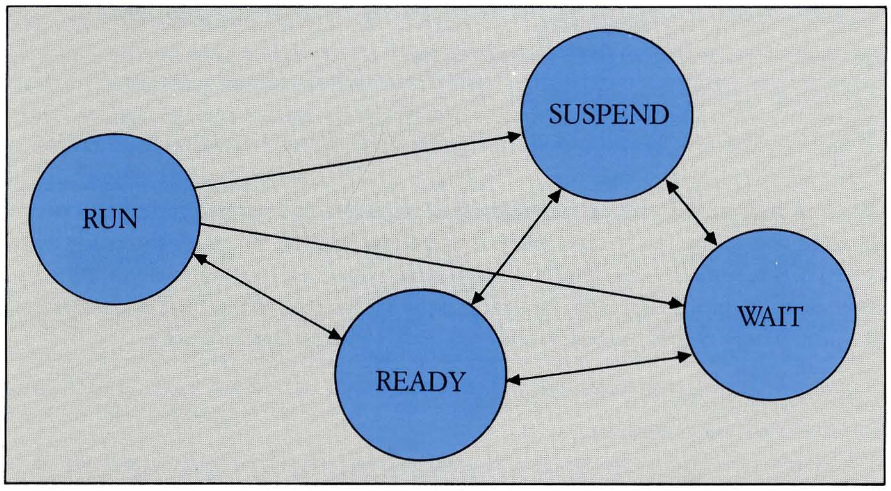

Figure 8. ***Process States.*** *Master processes and subprocesses are always in one of four states.*

RUN—A process in the run state has control of the CPU (i.e. it is the process currently executing).

READY—A process in the ready state is not running but is ready to run as soon as possible as determined by the scheduler (details on process scheduling are discussed later). All processes are initially in the ready state immediately following their creation.

WAIT—A process in the wait state is waiting for some specific set of conditions to be satisfied. It may be waiting for a particular amount of time to elapse, for the occurrence of a specific event or series of events, for the receipt of a message, and so forth. A process may put itself (and only itself) into the wait state by issuing either of the two wait operations; wait_any (wait for any of the listed conditions to be satisfied) or wait_all (wait for all of the listed conditions to be satisfied).

SUSPENDED—A process in this state is not eligible for execution (i.e. it cannot enter the ready state) until it is resumed by another process in the same job. A process can put itself or any other process in the same job into the suspended state with the suspend operation.

State transition, or changes from one state to another, describe the behavior of the system according to the following rules:

- The initial state of every process is ready
- When the wait (or blocking) conditions for a process are satisfied it enters the ready state
- When a suspended process is resumed, it re-enters its previous state; if the process was in the ready state when it was suspended, it re-enters the ready state when it is resumed. Note that if a suspended process was previously in the wait state, if all its blocking conditions have been satisfied as of the time it is resumed, it immediately enters the ready state without entering the wait state.

Job and Process Scheduling

VAXELN jobs and processes are all assigned priorities by the programmer. A high priority indicates that the job or process should be given preference over other jobs and processes of lower priority when it is ready to execute. A job (currently executing or not) is rescheduled when one or more of its processes enter the ready state and the job's priority is higher than the priority of the currently executing job. Within that job, the process with the highest priority is given control (enters the run state).

Job rescheduling, which is always pre-emptive—no round-robin or time slicing—is illustrated in the state diagram of Figure 9.

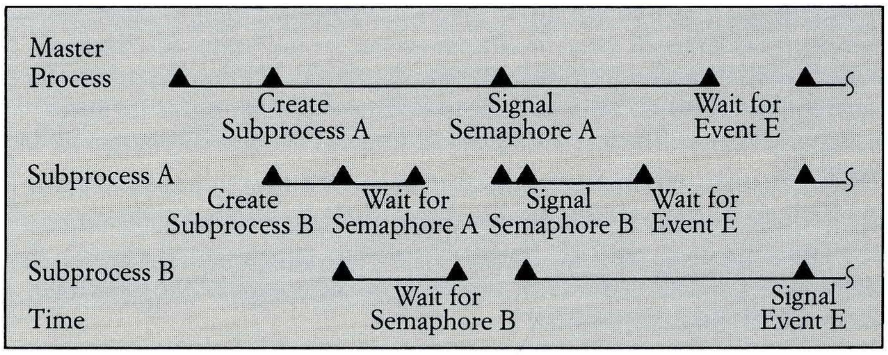

Figure 9. **Process Scheduling.** *A master process and subprocesses A and B make use of semaphore A and event E to synchronize execution.*

Jobs may be assigned 32 levels of priority (zero being the highest). Within jobs, processes may be assigned 16 levels of priority independent of their parent job's priority. Since processes are automatically rescheduled in a predictable way, the developer can design a system in which there are no important or noticeable delays in a program's execution, even though it spends at least some of its time idle while other programs execute. In principle, a program's execution speed is determined by the speed of the slowest thread of execution.

Job Virtual Address Space ☐ VAXELN uses the VAX memory management hardware to map jobs into individual and unique virtual address space. When a system is initially loaded onto the target VAX, the kernel maps the system image containing all the program, shareable runtime and kernel images into the system region (SØ) of the VAX virtual address space. The system region maps the system image and kernel data.

When a job is created to run a program image (process), the kernel creates a page table in the program (user) region (PØ) and maps the program image into the PØ region of the job's virtual address space. Each job's PØ region maps the program image, data and message buffers. The kernel also makes a copy of any read/write data in the program image, though no copy is made of read-only code or data. This means that if there are multiple jobs in a system running the same program, there is only one copy of the read-only code and data, and as many copies of the read/write data as there are jobs running the program.

The PØ region is used for static variables and message text. Since the runtime libraries use variables allocated in the PØ region for many of their data structures, the context of open file variables is also mapped into the PØ region.

All processes in a job share the same PØ page table and, consequently, the same PØ region. This means that any data in the job's PØ region are accessible to other processes in the same job. Assuming proper synchronization methods are used by the processes, a pointer to any variable in the PØ region can be passed to any other process within the job.

For each process created for a job, the kernel allocates a P1 page table and maps the process' kernel and user stacks in the P1 region. The stacks are used by programs for all process-local variables and the process' context (e.g. procedure call frames). The P1 region does not map any area of the program image; it is used exclusively for dynamic memory. The kernel stack is fixed in size and is used by kernel procedures and any programs that run exclusively in kernel mode. The user stack is dynamically expandable by the kernel. This feature is important in that it means programs may start out with a minimal stack that will grow when necessary without preallocating memory that might be wasted given a program's behavior in any particular situation.

Interjob Communication □ In a VAXELN application, every job has a unique and protected virtual address space. Within a single processor, the kernel separates each job's virtual address space using the VAX memory management hardware. Within a local area network, each job's virtual address space is separated by virtue of the fact the jobs exist in the physical memories of different target systems. To make the network movement of data between jobs the same in the non-network as well as in the network case, message passing is the only means of interjob communication in VAXELN systems.

Messages □ A message is recognized as a block of contiguous bytes of memory. As VAXELN utilizes the Ethernet network protocol, the maximum size of a message is also imposed. Because message passing is a key principle in VAXELN programming, the kernel was designed to make message-passing operations extremely efficient.

There are two means by which to implement message passing provided in VAXELN; via datagrams or via virtual circuits. A datagram is the traditional DECnet-VMS datagram—low overhead, fast but no acknowledgement of delivery. Creating a virtual circuit between jobs guarantees delivery of the message (via the Network Services Protocol—NSP available from the VAXELN Network Server) although it is somewhat slower due to the extra overhead entailed in the guaranteed delivery mechanism.

Messages are transmitted from and received by ports, which may be created dynamically by jobs and processes. To facilitate communication between jobs, message ports can be assigned names. A name can be created and deleted dynamically, may be up to 31 characters long, and be associated with a specific port value. Since each name is associated with a port, a program can look up a name and use the returned port value for communication with other jobs or processes.

Names are defined as being either local or universal. Local names are known only to jobs and processes on the system (node) on which they are created. Universal names are known by all nodes (and thus all jobs and processes) in the network. This ability to apply a logical name to a port allows the distribution of the application across multiple VAX nodes on a local area network essentially transparent to the designer/programmer of VAXELN systems.

Message Transmission □ Ports and messages can be used in two ways to transmit data, as mentioned previously:

- Datagrams — One process can obtain the value of a port anywhere in the system or in a different system running on a different Ethernet node.
- Virtual Circuits — Any two ports may be bound into pairs called "virtual circuits". Since messages are the only means of interjob communication, and since jobs can be located or relocated among several Ethernet nodes, the circuit is the preferred method of message passing in VAXELN applications.

To send a message from a job to another job within the same system, the VAXELN kernel does not physically move any data. It merely unmaps the message buffer's address from the sending job's virtual address space and maps it into the receiving job's address space. In the case where the communicating jobs are on different Ethernet nodes, the VAXELN Network Service physically transports the data across the network where it is placed into the receiving job's virtual address space.

Networking Part of the VAXELN Architecture

VAXELN utilizes the DECnet Data Access Protocol (DAP) in all communication scenarios within an application, not just message passing. Console and disk I/O for example, also utilize DAP as their highest-level interface. All VAXELN drivers have DAP front-ends to facilitate transparent multiprocessing in local area network configurations. Thus if a job or a disk file is moved off one processor and made resident on another node, the application program does not have to be modified at all; the inclusion of the supplied network server will assume the responsibility for ensuring the validity of the communication path. Thus, a program on one node may open, read and write files located on another system transparently to the application. This includes VMS nodes that may be physically connected to the local area network (as long as access is limited to sequential files).

VAXELN File Services

VAXELN provides the ability to create, read and write ODS-II, FILES-11 compatible disk files from the application. Files are limited to VMS sequential files (sequential and random access) via an RMS-compatible calling interface. Indexed, multikeyed, and relative access to RMS files is not supported. An optional layered product, Rdb/ELN (V1.2 or later), may be used to provide a full relational database facility for a VAXELN application.

It is worthy of note that VAXELN applications need not have any disks at all as part of the system; system images may be loaded into the target hardware over the Ethernet, from magnetic tape or from ROM as well as from disk. This capability is extremely important where a VAXELN target system may be in a physical environment where disks cannot be used, yet file access is necessary to the application. The disks may be located on a "safer" system elsewhere on the network, yet the isolated applications may still access file data as if the disks were physically located on the same system.

Performance Characteristics

VAXELN is designed for realtime applications, where responsiveness and predictability are crucial. The VAXELN kernel is highly optimized to take full advantage of the VAX architecture, and leaves as little as possible overhead between the application code and the hardware.

For example, in response to external events (interrupts), the VAXELN kernel imposes no more than three machine instructions from the time the interrupt is recognized by the processor until the first instruction in the application's interrupt service routine begins to execute. Table 3 represents actual timing data for VAXELN's performance as measured for interrupt latency, context switching (the time required to change the thread of execution from one process to another) and process synchronization (SEMAPHORE versus MUTEX). All timings are in microseconds and were achieved using a MicroVAX II processor.

Table 3 VAXELN Timing Data

Interrupt Latency	—	33 μsec
Context Switch	—	285 μsec
MUTEX (without context switch)	—	6 μsec
MUTEX (with context switch)	—	447 μsec
SEMAPHORE (without context switch)	—	289 μsec
SEMAPHORE (with context switch)	—	439 μsec

digital

Digital Equipment Corporation
Channels Marketing Group
4 Mount Royal Avenue
UP02-3
Marlboro, Massachusetts 01752

In Europe:
12 Avenue de Morgines
Case Postal 5103 CH-1213
Petit-Lancy 1
Geneva, Switzerland

In Canada:
DEC of Canada LTD.
P.O. Box 13000
Kanata, Ontario K26929

Printed in USA. EA29580-68/86 12 03 20.0 Copyright © 1986. Digital Equipment Corporation. All rights reserved.

102707474