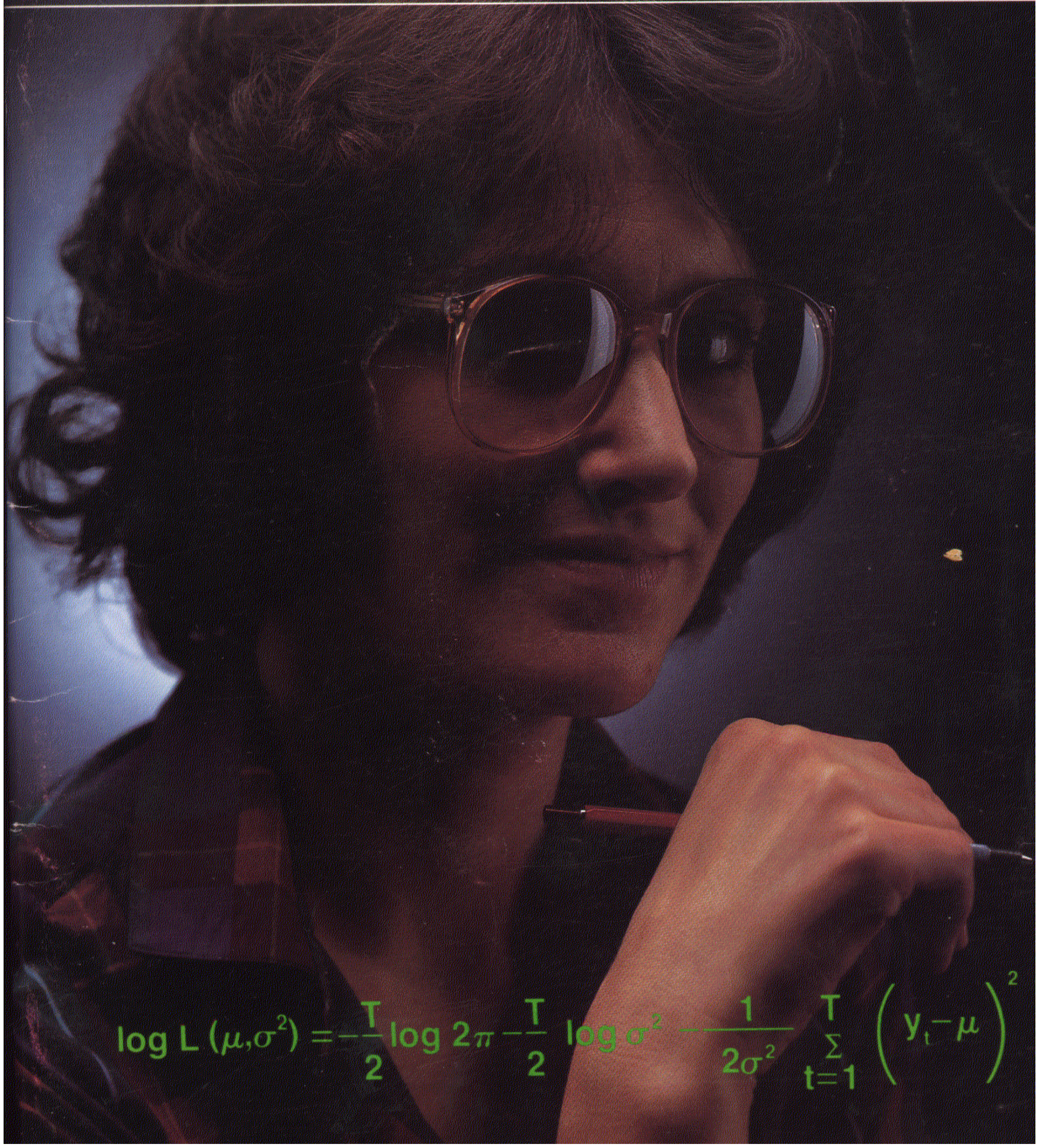
System Architecture

CONTROL DATA

CYBER 180 Computer Systems



$$\log L\,(\mu,\sigma^2) = -\frac{T}{2}\log 2\pi - \frac{T}{2}\log\sigma^2 - \frac{1}{2\sigma^2}\sum_{t=1}^{T}\left(y_t-\mu\right)^2$$

The CYBER 180s. Now accepting applications.

Performance. Security. Application portability. Compatible growth. The CYBER 180 computers have been designed for the 1990s and beyond to provide a comprehensive answer to these diverse computing requirements.

From the CYBER 180 Supermini 810 to the CYBER 180 Model 990, these systems offer a performance range of 1 to 60–the largest compatible growth path in the industry. And all provide security safeguards built into the hardware.

Through the unique architecture of the CYBER 180, it is possible to run two operating systems simultaneously–NOS (6-bit 60-bit words) and NOS VE (8-bit 64-bit words)–in the same memory and central processing unit (CPU).

The CYBER family offers easy application portability from other systems. And within the family all applications and systems software are portable without change.

And because they meet the widest range of user requirements–in particular, for response time, terminal loads and security measures–the CYBER 180s represent a major advance in applications performance.

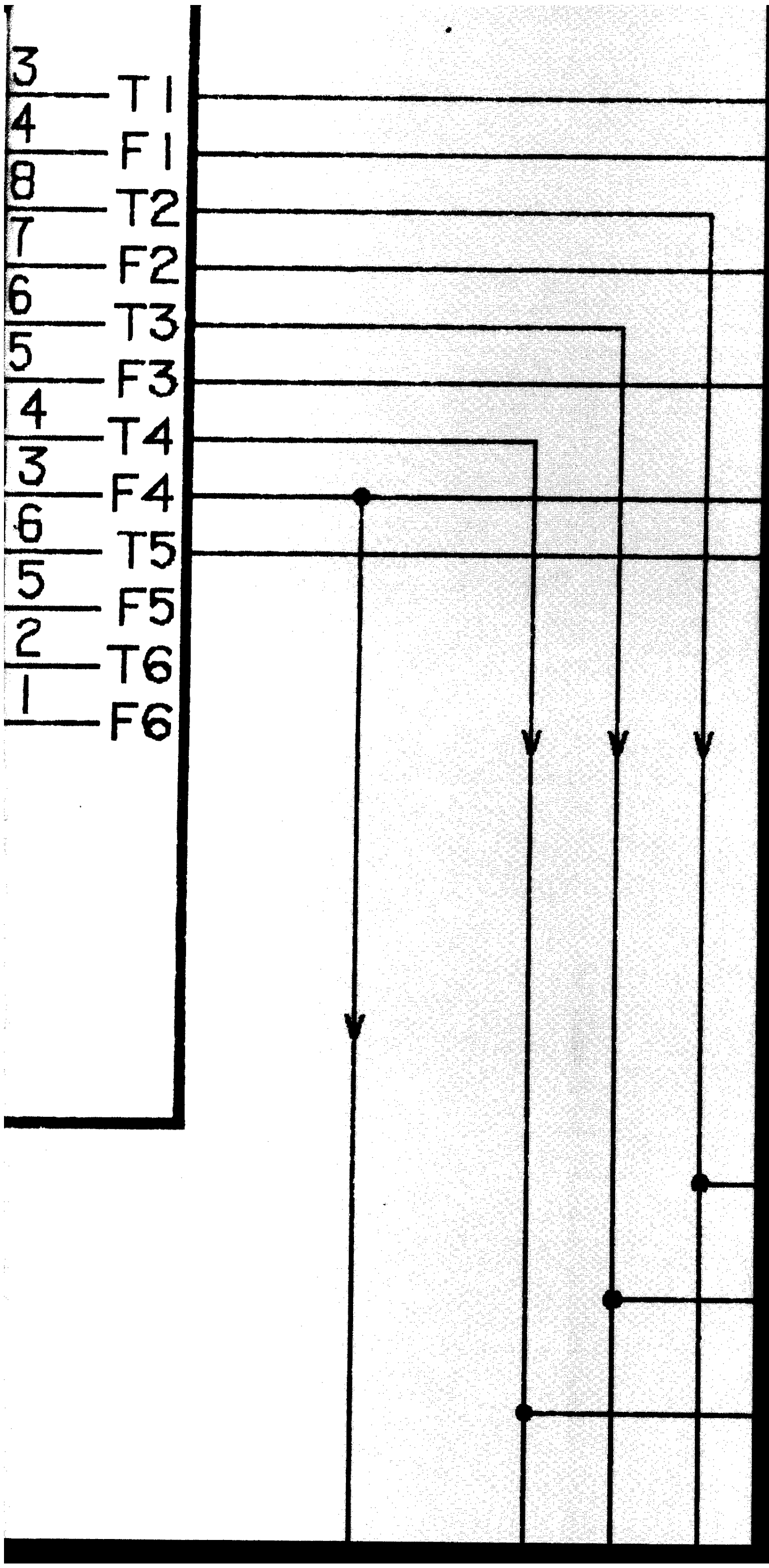
By running a wide variety of applications–both from Control Data as well as other software sources–the CYBER 180s provide cost effective solutions for manufacturing, energy development, education, research and government.

System Architecture

CONTROL DATA

CYBER 180 Computer Systems

Contents



Introduction to the Systems

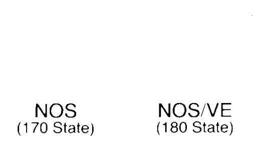
Introduction

This overview describes the hardware architecture of Control Data's CYBER 180 Series Computer Systems and the CYBER 170 Series models 815, 825, 835, 845, and 855 (hereafter referred to simply as the CYBER 180 series). The CYBER 180 series computer systems support two operating systems - NOS (the Network Operating System), an established, feature-rich system, and NOS/VE (the Network Operating System/Virtual Environment), an advanced system designed for the development of future applications.

A unique feature of these computer systems is their ability to execute two operating systems concurrently using the same processor. This mode of operation is called dual state. NOS executes in what is called 170 state. NOS/VE executes in 180 state. (Figure 1.)

Figure 1

Dual State



Many aspects of the hardware work the same regardless of the state in which the computer system is operating. Other aspects vary depending on the state to let each operating system take full advantage of particular hardware features. The next chapter describes the common parts of the computer systems. Subsequent chapters describe 170 state, 180 state, and finally, how the two work together, dual state.

But first, the remainder of this chapter discusses the general architectural concept behind all of these computer systems and how that concept is put into action.

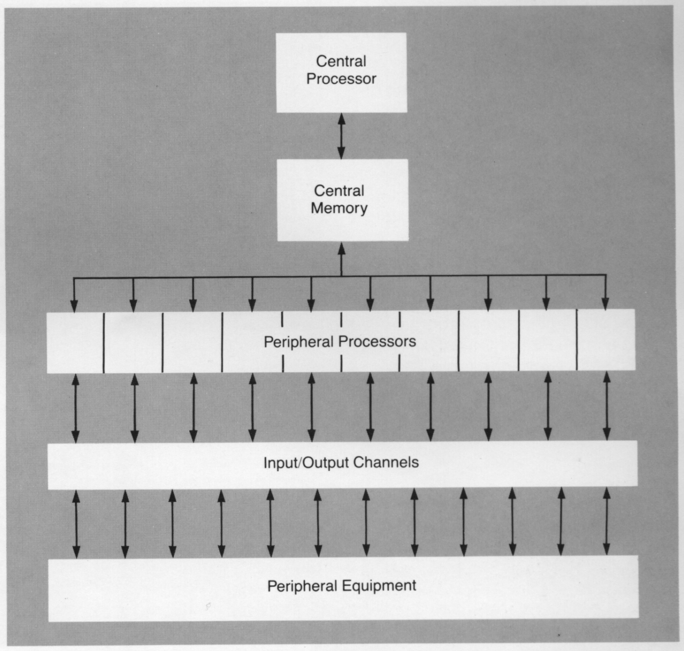
The Architectural Concept

The architecture of these computer systems exemplifies the concept that the processor should fit the process. Each model has a large central processor that handles all computation and execution of user programs. Smaller peripheral processors handle all input/output operations and some operating system functions, leaving the central processor free to work on user programs. This architectural concept accounts for the high system throughput evident in these computer systems in two ways.

First, the central processor is not burdened with input/output activities and minor operating system functions. Any time it would have spent doing these activities can be used to process user programs. Second, processing by the central processor and peripheral processors can be done in parallel. While input/output is being done by a peripheral processor, the central processor continues executing other programs; one does not have to wait for the other. Furthermore, because peripheral processors are independent, they also work concurrently to complete their separate activities.

Figure 2 shows the architectural concept of this hardware. The central processor performs most of the functions traditionally associated with the execution of user programs with the exception of the input/output. Central memory isolates the central processor from the peripheral processors and serves as the communication vehicle between them. Data is read from peripheral equipment by the peripheral processors and stored in central memory. It is only from central memory that the central processor accesses data and program instructions.

Figure 2



When additional data not currently in central memory is needed, the central processor temporarily stops the executing program and puts it in an inactive state while a peripheral processor gets the needed data. The central processor, however, does not remain idle. It begins execution of another program. When data is ready, the central processor is notified and can halt the new program and resume the interrupted program. Thus, many programs, in some state of execution, can be in the system at the same time.

Although only one program can use the central processor at a time, many programs may have peripheral processors performing operations for them.

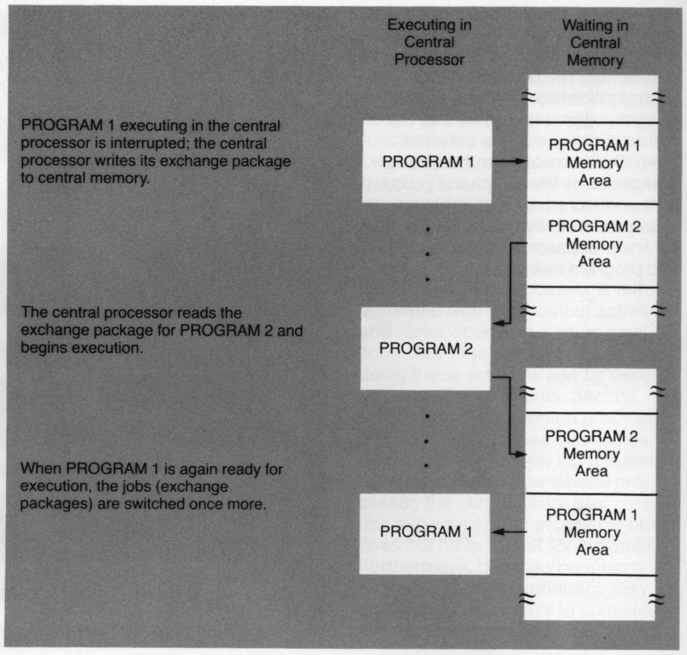
This process can be effective only if the method of switching programs in and out of execution is fast and efficient. This is made possible by the exchange jump operation and the exchange package.

The Method

For every program, an exchange package exists that contains all the information necessary to start or resume processing of the program. This information includes a description of the central memory used by the program, the next instruction word to be executed, the contents of all central processor registers, and the contents of certain status registers.

The exchange jump is the operation that actually switches programs. It is performed any time one program must be put in an inactive state and another program executed (for example, when additional data is needed or a high priority program comes along). Figure 3 illustrates the 170 state exchange jump. The 180 state exchange jump is similar in concept.

Figure 3

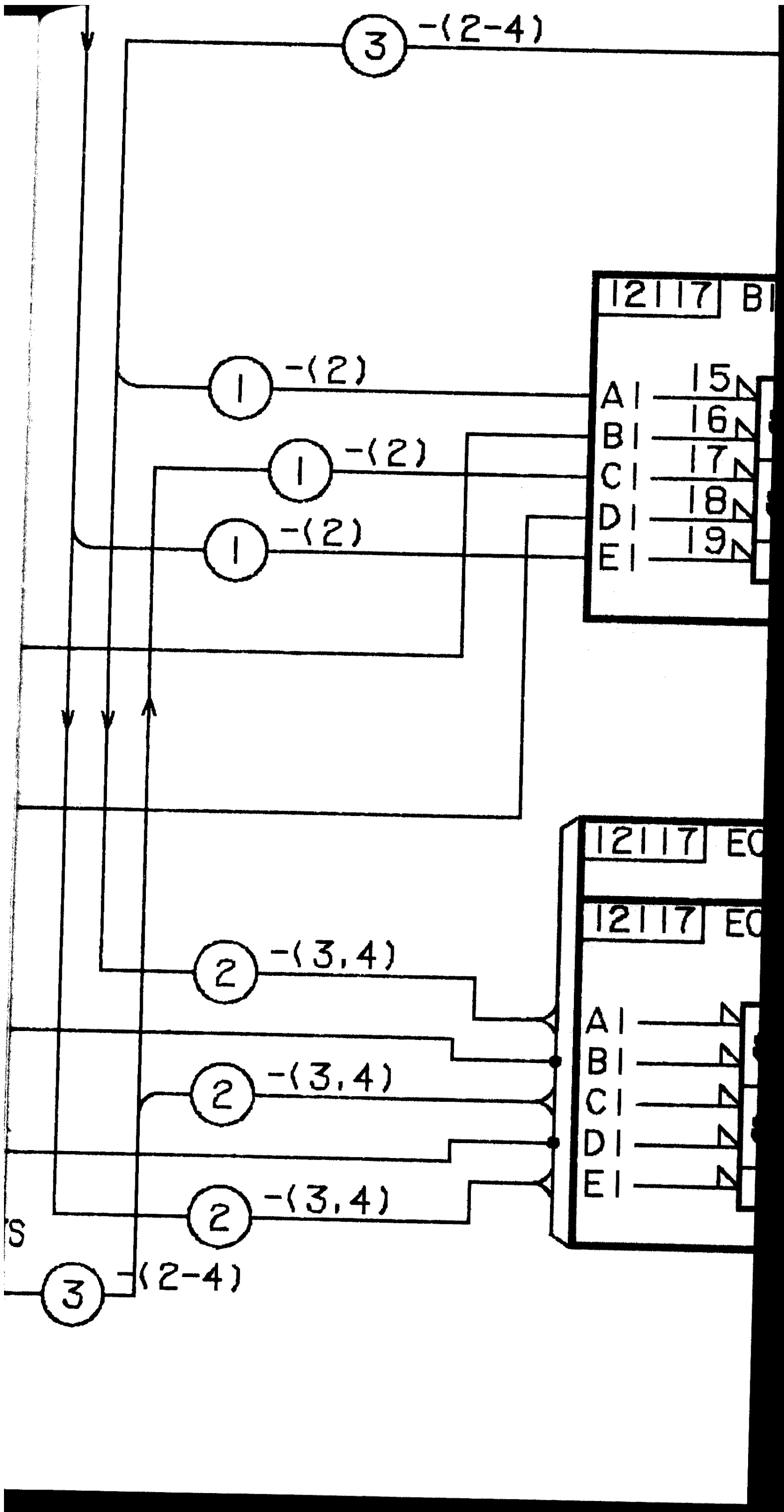

PROGRAM 1 executing in the central processor is interrupted; the central processor writes its exchange package to central memory.

The central processor reads the exchange package for PROGRAM 2 and begins execution.

When PROGRAM 1 is again ready for execution, the jobs (exchange packages) are switched once more.

The exchange operation is a fast and efficient method of switching programs. It is initiated and performed by a single instruction, and the exchange package it writes contains all the data necessary to start or resume a program. No additional operations to store registers are needed.

The concept of matching the processor to the process accounts largely for the high system throughput in the CYBER 180 series. It also allows a computational load to be balanced against an input/output load since the processors that perform each function are independent. A production environment with extensive input/output needs does not have to purchase a large central processor whose major function is computation just to obtain additional input/output power.



The Components of the Systems

Introduction

This chapter talks specifically about the individual components of the computer systems: the central processor, central memory, and the peripheral processors. It discusses the features that remain the same regardless of the state of operation, 170 state or 180 state.

The Central Processor

The central processor is a high-speed arithmetic processor dedicated to executing instructions of user and system programs. It performs computations in both fixed and floating-point arithmetic. The central processors increase in performance across the product line but within each state of operation they appear to the software to be common in areas such as:

- ☐ Registers
- ☐ Instruction set and instruction word formats
- ☐ Instruction lookahead
- ☐ Exchange jump operation and exchange package

Registers

The central processor contains a set of registers, some of which are used to hold data from instructions and others that contain information about the program itself. These registers are used by the central processor during execution and, as mentioned earlier in the first chapter, they are also the registers that are written into the program's exchange package if an exchange jump occurs.

To reduce the number of accesses to central memory for data and results, operating registers are used to hold operands and results during execution of a program. These operating registers allow a large degree of speed and latitude within the central processor. The central processor instruction set takes advantage of them and offers many register-to-register instructions. The number and type of registers varies depending on the state of operation and they are discussed in more detail in the later chapters on 170 state and 180 state.

To keep track of each program, support registers exist that contain information about the program. Again, the support registers vary depending on the state of operation and they are discussed later.

In either state of operation, the operating and support registers are copied to the program's exchange package should the program have to be interrupted and switched with another. These registers, along with some miscellaneous status registers, contain all the information needed to restart a program that has been interrupted. Both states of operation use an exchange package but the 170 state exchange package is a subset of the 180 state exchange package.

Instruction Set and Instruction Word Formats

In each state, there are two instruction sets: one for the central processor and one for the peripheral processors. The instruction set and instruction word formats for the central processor vary depending on the current state of operation. The peripheral processor instruction set is actually the same for both 170 state and 180 state; only the format of the word used changes.

The central processor instructions perform primarily arithmetic computations, data manipulation, memory transfers, and exchange jumps but no input/output operations. All input/output instructions are contained in the peripheral processor instruction set. The applications programmer, however, sees none of this separation between the central processor and peripheral processors, except as outstanding performance. The compilers, not the programmer, separate functions between the central processor and the peripheral processors. Each instruction set is discussed in more detail in the later chapters on 170 state and 180 state.

Memory

Instruction Lookahead

The instruction lookahead process is present on all models of the CYBER 180 series. It is based on the assumption that, in most cases, program instructions are located in consecutive central memory words. Lookahead hardware reads a certain number of instruction words beyond the currently executing instruction word and keeps them in reserve, usually in a set of registers that can be thought of as a first-in, first-out buffer. Then, when the current instruction has been executed, the next instruction is available immediately. Ideally, the lookahead hardware keeps the central processor continually supplied with instructions, and there is no waiting for a reference to central memory.

The number of instruction words read ahead and the method of handling branch instructions varies slightly from model to model. Keep in mind that one instruction word can hold up to four instructions.

The models on the lower end of the series read up to two instruction words ahead of the current instruction word but have no provision for branch instructions.

Intermediate models read up to three instruction words ahead of the current instruction word. If a conditional branch is detected, the central processor reads two words starting at the branch's destination address and holds them, in addition to the words from the regular program sequence, until the branch is resolved. If the branch is taken, those two words are available immediately; if not, they are discarded, and processing continues with the next word in the original three-word buffer. This can be thought of as a five-word instruction lookahead buffer with three words for the regular program sequence and two words for a conditional branch sequence.

The models on the upper end of the series read up to six instruction words ahead of the instructions contained in the current word in execution. If a conditional branch is detected, the central processor assumes the branch will be taken. The next instructions are read from the branch's destination address. If the branch is taken, the instructions are available immediately; if not, they are discarded, and processing and reading ahead are resumed at the instruction following the branch. Even when the branch is not taken, there is minimal delay because the nonbranch instructions are available from a high-speed cache memory.

Note that in all of these cases, the instruction lookahead buffer cannot be reread and, therefore, does not support program looping within the buffer. However, in machines on the upper end of the product line, the instructions are held in cache memory so program looping is generally done without references to central memory.

For extremely high performance, the model 990 uses advanced hardware techniques for lookahead that are quite different from those described here. See the Model 990 brochure (publication number 204,121) for more information on the model-dependent differences that provide that performance.

Central Memory

Word Size

The length of a central memory word is always 64 bits with an additional 8 bits used for error checking and correction. In 170 state, 60 bits are used with five 12-bit bytes to a word. In 180 state, 64 bits are used with eight 8-bit bytes to a word. In 180 state, memory can be addressed at both word and byte boundaries.

Addressing and Program Isolation in Memory

Likewise, the method of addressing and isolating programs in memory differs depending on the state of operation. NOS in 170 state uses real memory addressing on word boundaries only. NOS/VE in 180 state, on the other hand, uses virtual memory addressing on both word and byte boundaries. Both subjects are discussed later in the chapters on 170 state and 180 state.

Extended Memory

Extended memory is available only in 170 state and is described further in that chapter.

Peripheral Processors

The peripheral processors perform input/output operations and, in 170 state primarily, some operating system functions. Their major role is to do input/output, leaving the central processor almost entirely free for user programs. There is, of course, a central processor monitor, but the amount of time it spends in execution is very small compared with other vendors' systems.

Peripheral processors save considerable central processor time compared to other systems where the processor must perform all of the input/output activity rather than work on user programs. Even when that processor is not directly involved in the program, it must often monitor the proceedings and, thus, loses processing time that would otherwise be available for programs.

A comparison of the two types of systems might appear as shown in Figure 4.

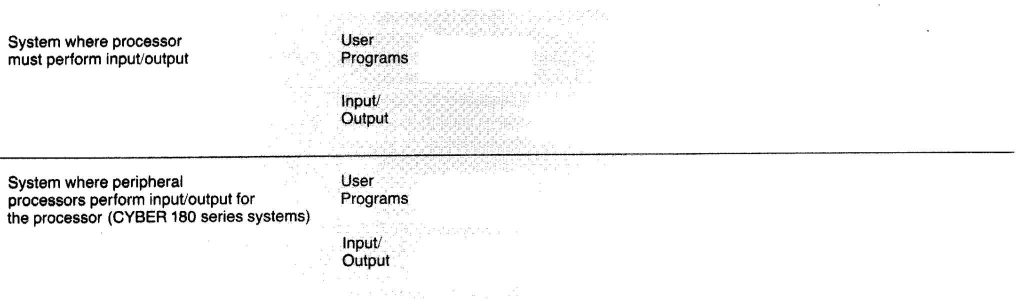
Of course, there is some overlap between performance of system functions, such as input/output, and pure program processing time on the other types of systems but not nearly as much as is evident on the CYBER 180 series. In addition on these machines, each peripheral processor operates independently; therefore, many peripheral processors can be performing operations for many programs at the same time.

The central processor cannot, except by software protocol, call a peripheral processor. This in itself forms the basis for high system security and program integrity.

Peripheral processors are logically independent but physically contained in one unit - the input/output unit. The minimum number of peripheral processors available is 10, but this number can be expanded to 20. Peripheral processors on all models of the CYBER 180 series are functionally identical although some features vary depending on the state of operation. For example, the length of a peripheral processor memory word is 16 bits but the actual number of bits used varies depending on the current state of operation. Within a particular state, peripheral processors share such features as:

- ☐ Memory and word size
- ☐ Method of reading and writing central memory words
- ☐ Instruction set
- ☐ Registers
- ☐ Channel usage

Figure 4


System where processor must perform input/output
User Programs
Input/ Output
System where peripheral processors perform input/output for the processor (CYBER 180 series systems)
User Programs
Input/ Output

Memory and Word Size

A peripheral processor has a memory of 4096 words. The length of a peripheral processor memory word is 16 bits. In 170 state, the least significant 12 bits of the 16-bit word are used. In 180 state, the full 16 bits are used. Regardless of the word length used in a particular state, the peripheral processor uses the same method to read and write words to and from central memory.

Reading and Writing Central Memory Words

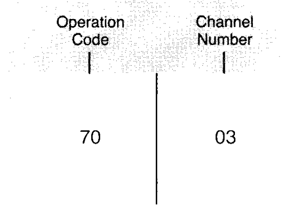
Central memory words are read and written between central memory and peripheral processor memory by the peripheral processors using a hardware buffering mechanism that assembles and disassembles the words. When a central memory word is read, it is disassembled into a fixed number of peripheral processor words and sent to successive locations in the peripheral processor's memory. To write a central memory word, the hardware assembles several successive peripheral processor words into one larger word and sends it to central memory. An obvious advantage of this process is that there is only one access to central memory rather than several. In 170 state, five 12-bit peripheral processor words are assembled and disassembled to transfer 60-bit central memory words. In 180 state, four 16-bit peripheral processor words are assembled and disassembled to transfer 64-bit central memory words. (The hardware that performs this function is called a read/write pyramid.)

Instruction Set

The instruction set used by the peripheral processors is the same regardless of the state of operation. The instruction set emphasizes input/output operations and system monitoring activities. Peripheral processors address only real memory and, therefore, need not change because of virtual memory considerations in 180 state.

Peripheral processor instructions are either one or two peripheral processor words long. Of the 70 instructions in the set, 43 are single-word instructions, and 27 are double-word instructions. Many instructions require only a 2- or 3-digit operation code and a single operand. For example, the following instruction (Figure 5) transfers a word from channel 3 to a peripheral processor register (the A register).

Figure 5



The instruction set includes instructions to perform:

- ☐ Input/output
- ☐ Testing and setting of channel flags
- ☐ Read and write operations between central memory and peripheral processor memory
- ☐ Arithmetic and logical operations
- ☐ Conditional and unconditional jumps
- ☐ Central processor exchange jumps

Registers

The peripheral processor registers are the same in both 170 and 180 state. Each peripheral processor has its own set of registers that hold information necessary for executing the processor's current instruction. One of these registers holds the program address; others can hold input/output data words, functions, operands, and addresses used in executing instructions.

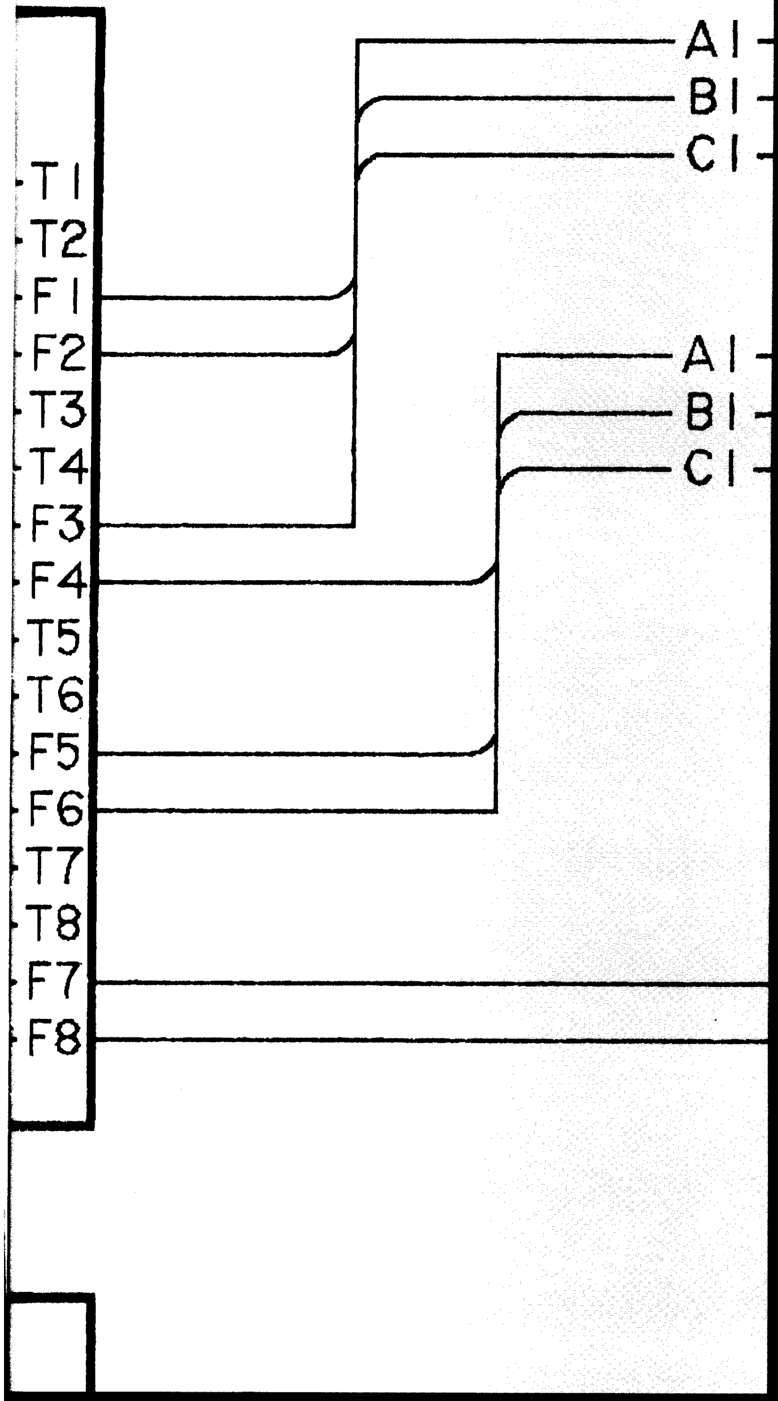
Two registers in particular, the A and R registers, work together to form large central memory addresses for read/write instructions. The A register holds one operand; this operand may be a central memory address. However, this 18-bit register is not always sufficient to hold the large central memory addresses available on the CYBER 180 series. When necessary, a relocation register, the R register, is used to supplement the A register to form a larger address.

Channel Usage

Peripheral processors communicate with each other and with external devices (peripheral equipment) over input/output channels. One or more of these devices can be connected to a channel and any peripheral processor can access any channel assigned to its particular state (170 or 180). The number of channels available can be as many as 24 depending on the model.

Channel instructions in the peripheral processor instruction set direct all activities related to peripheral equipment. They select a specific piece of equipment on a channel and transfer data to or from that equipment. Although more than one piece of equipment can be connected to a channel, one device at a time uses the channel to transfer data.

170 State



Introduction

The CYBER 180 series executes NOS in a mode known as 170 state. In this state, the hardware uses features that optimize NOS performance. It operates using real memory addressing on word boundaries. This chapter describes the features and characteristics that are unique to 170 state.

170 Central Processor

The central processors vary in performance from model to model but in 170 state they appear to the software to be common in areas such as:

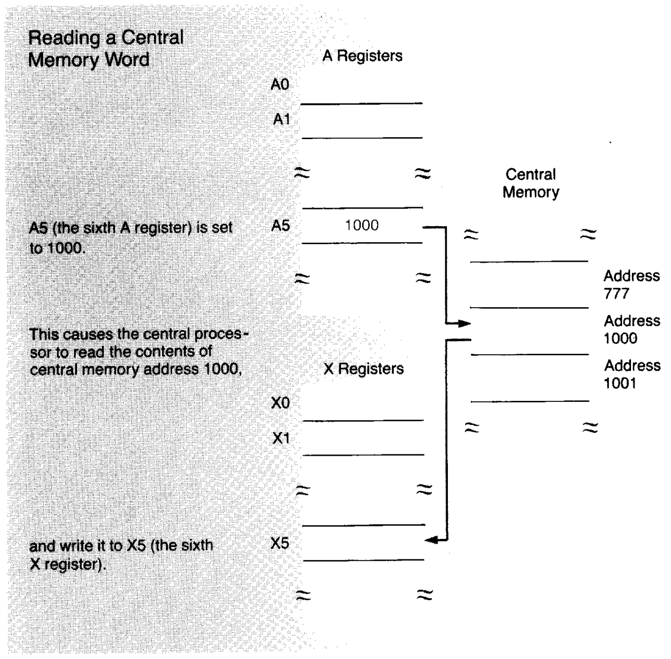
- ☐ Registers
- ☐ Instruction set and instruction word formats
- ☐ Exchange jump operation and exchange package

Registers

As was mentioned earlier, the central processor contains a set of registers which are used to hold data from instructions and to contain information about the program itself. These registers are used by the central processor during execution and they are also the registers that are written into the program's exchange package if an exchange jump occurs.

Operating registers are used to hold operands and results during execution of a program. In 170 state, there are eight 60-bit X registers that hold the operands. (A central memory word in 170 state is also 60 bits long.) The eight X registers are numbered X0 through X7 (where X0 is the first register and so on). Eight 18-bit A registers hold the central memory addresses of the operands.

Figure 6


Reading a Central Memory Word
A Registers
A0
A1
A5 (the sixth A register) is set to 1000.
A5
1000
Central Memory
Address 777
Address 1000
Address 1001
This causes the central processor to read the contents of central memory address 1000,
X Registers
X0
X1
and write it to X5 (the sixth X register).
X5

The eight A registers are likewise numbered A0 through A7 (where A0 is the first register). The X and A registers work together to read from and write to central memory.

To read a central memory word (Figure 6), one of the registers A1 through A5 is set to a central memory address. The central processor accesses that address and writes its contents in the corresponding X register (X1 through X5). There is no explicit read memory instruction; instead, the set A register instruction performs that function.

To write a central memory word (Figure 7), A6 or A7 is set to a central memory address. The central processor transfers the word in the corresponding X register (X6 or X7) to the central memory address.

Of course, instructions also exist to move the contents of one register to another. All 60-bit operands involved in computation are taken from and returned to the X registers.

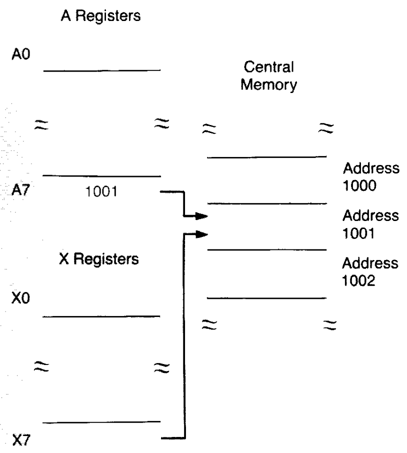
There are also eight 18-bit B registers used primarily for indexing. They can be used, for example, to increment and decrement program loop counts.

These 24 operating registers allow a large degree of speed and latitude within the central processor. The central processor instruction set takes advantage of them and offers many register-to-register instructions.

To keep track of each program, support registers exist that contain information about the program. (Figure 8.) The RAC register holds the reference address of the program. This is simply the starting address of the program in central memory. The FLC register holds the length of the program in central memory (its field length). These two registers define

Figure 7

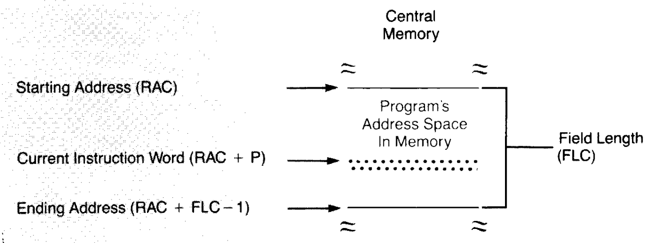
Writing a Central Memory Word



A7 (the eighth A register) is set to 1001.

This causes the central processor to write the contents of X7 (the eighth X register) in central memory address 1001.

Figure 8


Central Memory
Starting Address (RAC)
Current Instruction Word (RAC + P)
Ending Address (RAC + FLC − 1)
Program's Address Space In Memory
Field Length (FLC)

the entire user address space of a single program. The contents of the RAC register added to the contents of the FLC register provides the ending address of the program in central memory. (RAC + FLC -1 is the actual address.) Any attempt to go outside these boundaries causes a hardware interrupt to occur.

Within the program itself, all addresses are given relative to a base address of zero, the beginning of the program. This method allows programs to be loaded anywhere in the system simply by changing the RAC register; no internal program addresses are changed because they are still given relative to the base address of zero.

The P register is the program address register. It is the address within the program of the instruction word currently being executed. It is relative to the beginning address of the program which is always zero. Thus, to find the absolute central memory address of the instruction to be executed, hardware adds the contents of the RAC register to the contents of the P register.

All of the preceding registers, both operating and support registers, are copied to the program's exchange package should the program have to be interrupted and switched with another. These registers, along with some miscellaneous status registers, contain all the information needed to restart a program that has been interrupted.

Instruction Set and Instruction Word Formats

The central processor instructions perform primarily arithmetic computations, data manipulation, memory transfers, and exchange jumps but no input/output operations. All input/output instructions are contained in the peripheral processor instruction set.

In 170 state, the central processor uses the least significant 60 bits of the 64-bit word and leaves the other 4 bits unused. Central processor instructions are 15 bits, 30 bits, or 60 bits long. An advantage of this central processor word size is that 15- and 30-bit instructions can be packed into it, allowing as many as four 15-bit instructions to be stored in one central processor word. This obviously saves space and time (only one central memory access is required instead of four). Because of the instruction lookahead discussed earlier, efficient programming can dramatically increase the number of available instructions awaiting execution in the central processor. With few restrictions, almost any combination of 15- and 30-bit instructions are allowed in one word.

The central processor instruction set is extensive, consisting of 85 instructions, yet simple in structure. It is powerful but easy to learn and easy to use. Of the 85 central processor instructions, 55 are 15-bit instructions and 27 are 30-bit instructions; only 3 instructions require the full 60 bits. Many instructions require only the operation code (a 2- or 3-digit number) and the registers containing the operands and results.

The instruction set features floating-point add, multiply, and divide instructions with double and single precision, and rounded and unrounded results. In addition, there are instructions to:

- ☐ Perform integer arithmetic (add, subtract, multiply, and divide)
- ☐ Pack and unpack floating-point numbers
- ☐ Normalize floating-point numbers
- ☐ Set the X registers (data), A registers (addresses), and B registers (indexes)
- ☐ Perform logical operations, transmission operations, and shift operations
- ☐ Perform conditional, unconditional, and return jumps
- ☐ Perform exchange jumps, as described in the first chapter
- ☐ Read and write single words to and from central memory or extended memory
- ☐ Transfer blocks of data between central memory and extended memory

Operands are represented in one's-complement form. Operands can be 18 or 60 bits long.

Because of the simplicity of this instruction set, the central processor assembler can be learned in a relatively short time. Furthermore, the assembly language programmer need not understand the peripheral processor input/output architecture to code the central processor. All input/output is initiated with simple macros. Of course, for those environments that demand it, a peripheral processor instruction set and assembler are also available.

170 Memory

Central Memory

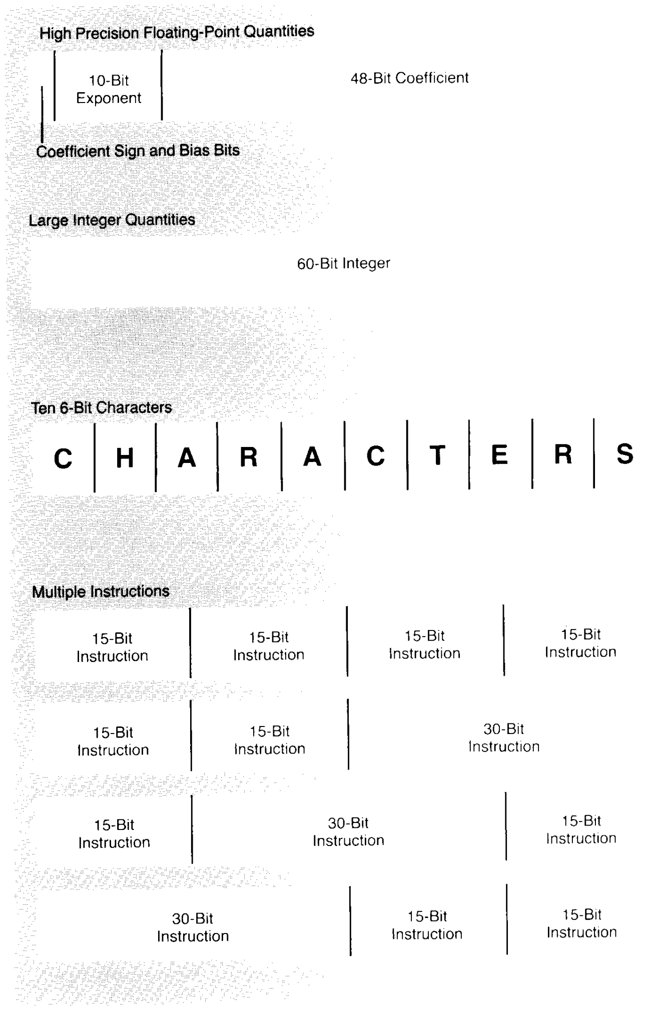
Word Size

In 170 state, central memory words are 60 bits long with additional bits used for error checking and correction. This data word size offers several advantages (Figure 9):

- High precision floating-point quantities
- Large integer quantities
- Ten 6-bit characters per word in a single word access from central memory
- More than one instruction per word because 15- and/or 30-bit instructions can be packed in one word

The bits reserved for error checking and correction are used for parity bits and codes that enable single-bit error correction and double-bit error detection (SECDED).

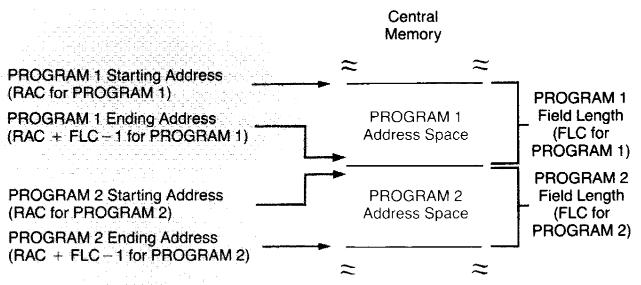
Figure 9



Addressing and Program Isolation in Memory

The RAC (reference address) and FLC (field length) registers that were described earlier in this chapter are used in the central processor to designate the beginning and ending addresses of the program in central memory. Each program in central memory has an RAC and FLC associated with it that separates and isolates it from all other programs. No program can read or write in another program's field length, either accidentally or deliberately. (Figure 10.)

Figure 10


Central Memory
PROGRAM 1 Starting Address (RAC for PROGRAM 1)
PROGRAM 1 Ending Address (RAC + FLC − 1 for PROGRAM 1)
PROGRAM 2 Starting Address (RAC for PROGRAM 2)
PROGRAM 2 Ending Address (RAC + FLC − 1 for PROGRAM 2)
PROGRAM 1 Address Space
PROGRAM 2 Address Space
PROGRAM 1 Field Length (FLC for PROGRAM 1)
PROGRAM 2 Field Length (FLC for PROGRAM 2)

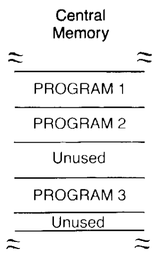
Hardware aborts any program that attempts to access data outside of the limits of its address space (as defined by RAC and FLC). Only the operating system can access data in any program. Communication between address spaces (programs) can be done only via the operating system and always using a strictly enforced software protocol. Memory is not shared between user programs.

As far as the hardware is concerned, RAC and FLC are limited only by the physical size of memory. However, system software can impose restrictions on RAC and FLC that the hardware cannot override. Individual programs are limited to 131,072 words of central memory. In addition, certain areas of central memory are reserved for the operating system only, although everything else is considered available for programs. Software can also limit the number of individual programs in central memory. NOS allows up to 33 programs in central memory at any one time. Because a program must be in central memory to be eligible for execution (the central processor communicates only with central memory), this means a maximum of 33 programs can be in central memory in some state of execution at the same time. However, the total number of programs in the system is limited only by disk space. A much higher number could be waiting in swapped-out files in some state of execution or waiting in input queues for initiation.

RACs and FLCs for programs in central memory are not fixed. They are assigned by the operating system and may be changed according to the physical space available and individual program requirements. Within the program, of course, addressing always begins at address zero. This allows absolute binaries to be moved freely about memory because no internal addresses have to be changed when the RAC changes.

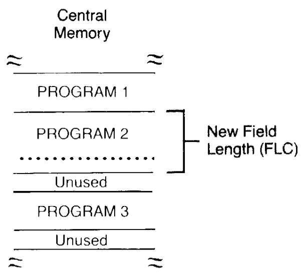
RACs change during the processing of a program if space needs to be reallocated for new programs or other programs' changing requirements. After the system has been running for a while, it's not unusual for gaps to develop between programs in central memory - as old programs complete and new ones begin. (Figure 11.)

Figure 11



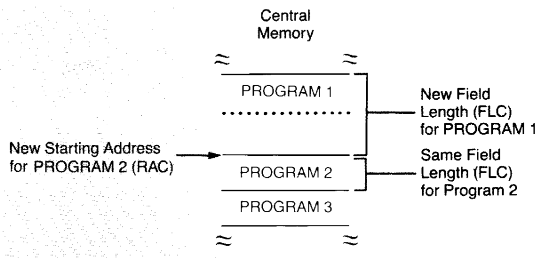
If the second program in Figure 11 (PROGRAM 2) needs more storage, it is an easy matter to take the unused area just below it. The field length is changed to include this space but no other values are changed. (Figure 12.)

Figure 12



If, however, the first program (PROGRAM 1) needs more storage, PROGRAM 2 must be moved. The system shifts PROGRAM 2 and adds the additional storage to PROGRAM 1. (Figure 13.)

Figure 13


Central Memory
PROGRAM 1
New Field Length (FLC) for PROGRAM 1
New Starting Address for PROGRAM 2 (RAC)
PROGRAM 2
Same Field Length (FLC) for Program 2
PROGRAM 3

These moves are called storage moves. They are completely transparent to users and do not affect the relative addresses within programs in any way.

As mentioned before, individual programs are limited to 131,072 words of central memory. If more storage is needed, extended memory is available.

Extended Memory

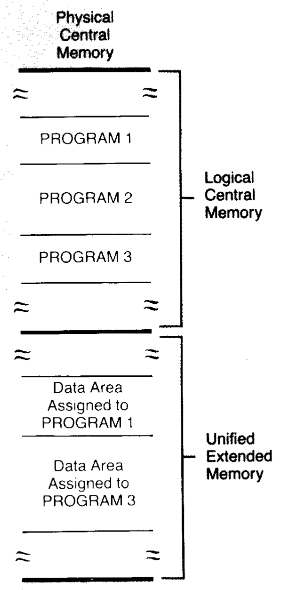
Extended memory is used for its large storage capacity and high transfer rates (faster than other mass storage devices).

- ☐ If additional memory is needed for data storage, extended memory can be used. For example, Control Data's FORTRAN allows a large array type to be declared for data, and this array can be in extended memory. (This extension to the ANSI 77 standard is referred to as LEVEL 2 data residency. COBOL also supports this extension.) Thus, data is limited only by the size of extended memory.
- ☐ Extended memory offers additional storage for operating system routines that must be accessed quickly and frequently, thus saving space in central memory for user programs and increasing system throughput.
- ☐ Files, such as user files, can also be declared resident in extended memory for fast access, thus saving access time to normal mass storage (disk) devices.
- ☐ In the larger systems, extended memory is also used for disk file buffering.

Central processor instructions are available to read and write single words and perform high-speed transfers of large blocks of data between central memory and extended memory.

Extended memory is available in 170 state by dividing physical central memory into two areas: one used for normal central memory purposes and one used for extended memory purposes. (Figure 14.)

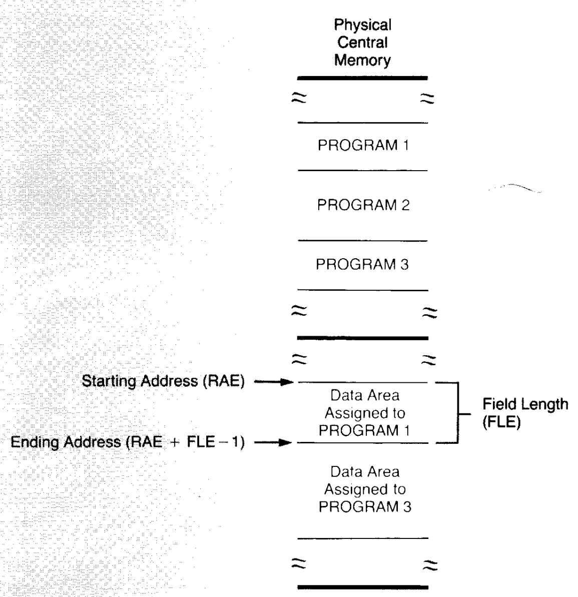
Figure 14



This partitioning is determined by the operating system software; there is no physical distinction. This type of extended memory is particularly useful for cases where additional program and data storage is needed.

When a program requests extended memory, a block is assigned for the duration of the program. In 170 state, this block is designated by two central processor registers, RAE (reference address, extended memory) and FLE (field length, extended memory). Notice the resemblance to the RAC and FLC registers that designate the program's central memory block. The RAE and FLE registers are also included in the exchange package when one program is exchanged for another. They also serve to separate and isolate the area in extended memory so that no other program can read or write into that area. (Figure 15.)

Figure 15


Physical Central Memory
PROGRAM 1
PROGRAM 2
PROGRAM 3
Starting Address (RAE)
Ending Address (RAE + FLE − 1)
Data Area Assigned to PROGRAM 1
Field Length (FLE)
Data Area Assigned to PROGRAM 3

170 Peripheral Processors

When a program enters the system, peripheral processors in 170 state read it, interpret commands, assign priority, assign resources, and then schedule it for central processor time. The central processor does none of these things; the peripheral processor does as much as possible to prepare the program for execution before the central processor receives it.

Peripheral processors not only perform these functions, but also monitor them. One peripheral processor is dedicated as the system monitor that controls the operating system and assigns activities to the other peripheral processors. Another dedicated peripheral processor contains the driver for the system display console. It is responsible for providing, through displays, information about all programs in the system, equipment, memory contents, and subsystems, to name a few. All this is done completely without interruption of the central processor. In addition, the system display driver sends operator commands to the system and displays messages from operating system routines.

Peripheral processors on all models of the CYBER 180 series are functionally identical. The peripheral processor instruction set, registers, and channel usage are the same for both 170 and 180 state and are described more fully in the preceding chapter on the system components.

In 170 state, peripheral processors share such features as:

- ☐ Memory and word size
- ☐ Method of reading and writing central memory words

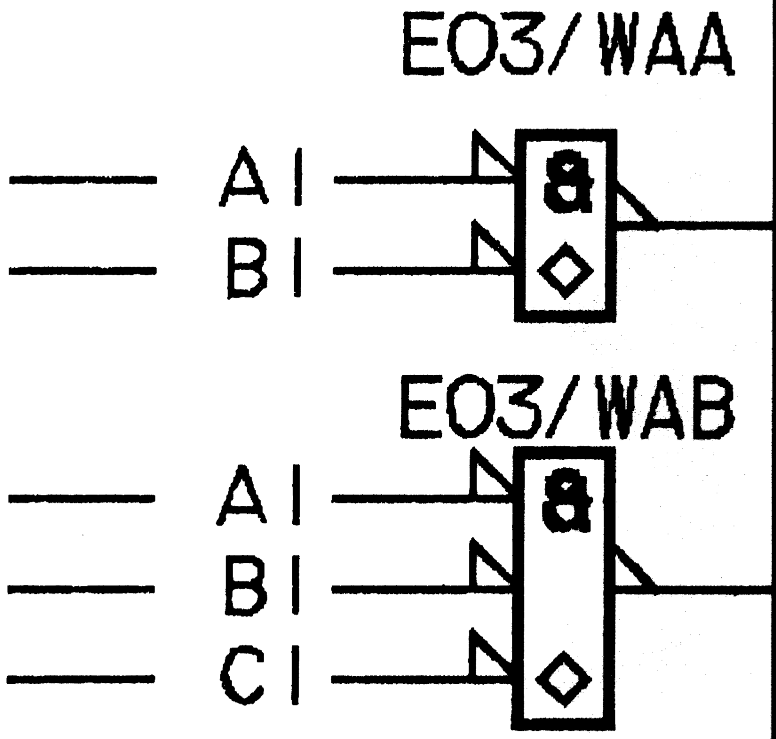
Every peripheral processor can communicate with any channel or other peripheral processor that is assigned to 170 state.

Memory and Word Size

In 170 state, a peripheral processor uses the least significant 12 bits of the 16-bit memory word leaving 4 bits unused. A hardware error occurs if an attempt is made to use those 4 bits while in 170 state.

Reading and Writing Central Memory Words

In 170 state, 60-bit central memory words are read and written by the peripheral processors using a hardware buffering mechanism that assembles and disassembles the words in 12-bit quantities. When a 60-bit central memory word is read, it is disassembled into five 12-bit words and sent to successive locations in the peripheral processor's memory. To write a central memory word, the hardware assembles five successive 12-bit peripheral processor words into one 60-bit word and sends it to central memory. An obvious advantage of this process is that there is only one access to central memory rather than five.


E03/WAA
A
B
&
◇
E03/WAB
A
B
C
&
◇

Introduction

The 180 series executes NOS/VE in a mode known as 180 state. In this state, the hardware uses features that optimize NOS/VE performance. It operates using virtual memory addressing on either byte or word boundaries. This chapter describes the features and characteristics that are unique to 180 state.

180 Central Processor

In 180 state, the central processors are common in the areas of:

- Registers
- Instruction set and instruction word formats
- Exchange jump operation and exchange package

Registers

Although the names and exact functions of registers in 180 state differ from 170 state, their underlying purpose is the same: to hold data from instructions and to contain information about programs. These registers are used by the central processor during execution and they are also the registers that are written into the program's exchange package if an exchange jump occurs.

To minimize memory references for arithmetic operands and results, there are 32 operating registers accessible to programs. These operating registers hold operands and results during execution of programs.

In 180 state, there are sixteen X registers that hold operands and can also be used for indexing. These registers replace the eight X registers and eight B registers present in 170 state. Each X register is 64 bits long. (A central memory word in 180 state is also 64 bits long.) The sixteen X registers are numbered, using hexadecimal notation, X0 through XF (where X0 is the first register and so on).

Sixteen 48-bit A registers hold the central memory addresses of the operands. They are given in the form of process virtual addresses which are discussed later in this chapter under 180 Memory. The sixteen A registers are likewise numbered A0 through AF (where A0 is the first register). There are more A registers available in 180 state than 170 state, sixteen as opposed to eight. Unlike 170 state, operations on A registers have no effect on the X registers; operands can be loaded into or stored from any X register.

The two instructions shown in Figures 16 and 17 transfer a word between a specified X register and central memory. In both cases, the address of the word transferred is the sum of eight times Q, giving the displacement, plus the byte number field from the specified A register.

The first instruction loads the specified X register (in this case, X4, the fifth X register) with the contents of the word specified by the A register (A5, the sixth A register). There is no offset because the Q field is zero.

The second instruction stores the contents of the specified X register (in this case, XE, the fourteenth X register) in the word specified by the A register (A6, the seventh A register) displaced by eight times Q. (This displacement means it is left-shifted three bits to become a word displacement.)

Figure 16

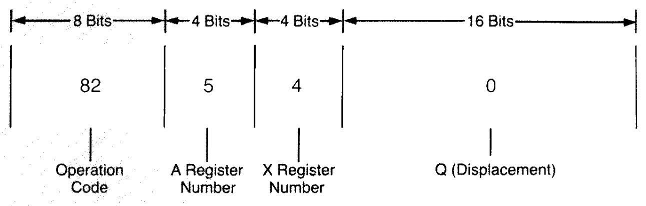
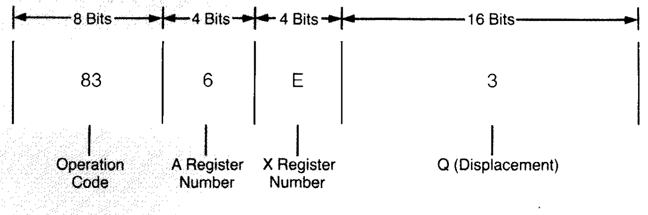


Figure 17



Additional registers keep track of each program. For example, the 64-bit P register is the program address register. It contains the address within the program of the instruction word currently being executed.

All of the preceding registers are copied to the program's exchange package if the program has to be interrupted and switched with another.

Besides the registers that contain information specific to a program, there are also registers that contain information about the processor itself, such as processor-dependent characteristics or processor interrupts. They monitor and report certain conditions concerning the hardware. Still others contain pointers to tables and exchange packages in central memory.

Instruction Set and Instruction Word Formats

The central processor instructions perform primarily arithmetic computations, data manipulation, memory transfers, and exchange jumps but no input/output operations. All input/output instructions are contained in the peripheral processor instruction set. In addition to the full set of central processor instructions shared by every model, the model 990 also executes vector instructions.

In 180 state, the central processor uses 64 bits for data in a word with additional bits used for error checking. Central processor instructions vary in length by byte multiples. Instructions less than a full 64-bit word can be packed together. For example, two 16-bit instructions and one 32-bit instruction can be stored in a central processor word, as can four 16-bit instructions.

The central processor instruction set includes instructions to:

- ☐ Perform integer arithmetic (add, subtract, multiply, and divide)
- ☐ Perform floating-point arithmetic (add, subtract, multiply, and divide)
- ☐ Move, edit, and translate character strings
- ☐ Perform packed and unpacked decimal arithmetic
- ☐ Load or store fields on 64-bit word, byte, and bit boundaries
- ☐ Extract or insert bit strings of 1 through 64 bits
- ☐ Load or store multiple A registers (addresses) and X registers (operands)
- ☐ Normalize floating-point numbers
- ☐ Perform logical operations
- ☐ Perform shift operations
- ☐ Perform conditional branch tests
- ☐ Perform exchange jump operations
- ☐ Perform call and return operations

In addition to these instructions, the model 990 also supports vector processing instructions. They provide for:

- ☐ Integer vector arithmetic
- ☐ Integer vector compare
- ☐ Logical vector arithmetic
- ☐ Integer/floating-point vector conversion
- ☐ Floating-point vector arithmetic
- ☐ Circular shift
- ☐ Merge, scatter, and gather operations with fixed stride
- ☐ Summation of a floating-point vector

All operands are represented in two's-complement form. Operands are 64 bits long with 32-bit integer operands used for memory addressing. Although both words and bytes can be addressed in 180 state, any word-oriented instructions or 64-bit operand arithmetic always use a word boundary; the 64-bits are not split among bytes of two different words.

The usage of most of these instructions is evident from the short descriptions above. The call/return mechanism, however, is a major feature of 180 state and warrants more of an explanation. This is given in a separate section later in this chapter called The Call/Return Mechanism.

Debugging Facility

The central processor also provides hardware debugging capabilities for programmers debugging programs in 180 state. It provides an interrupt capability during instruction execution. Actually, the instruction is not executed until the debug processing is complete. Users can select debugging based on a number of conditions:

- ☐ Whenever data is read from a specified area in virtual memory
- ☐ Whenever data is written into a specified area in virtual memory
- ☐ Whenever an instruction is fetched from a specified area in virtual memory
- ☐ Whenever a branch is made to a specified area in virtual memory
- ☐ Whenever a call instruction is issued to a procedure in a specified area in virtual memory

For any instruction issued, users can debug on any combination of these five conditions for up to 32 different areas in virtual memory.

180 Memory

Central Memory

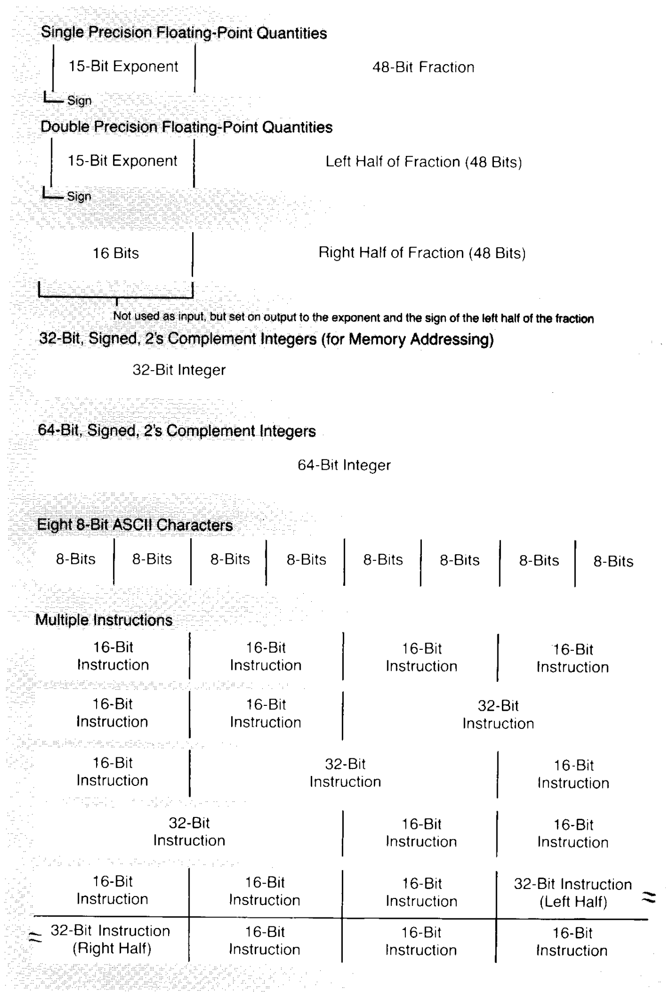
Word Size

In 180 state, central memory uses 64-bit words (eight 8-bit bytes to a word) with additional bits used for error checking and correction. This data word size offers several advantages (Figure 18):

- High precision floating-point quantities
- Large integer quantities
- Eight 8-bit ASCII characters per word in a single word access from central memory
- More than one instruction per word when 16- and/or 32-bit instructions are packed in one word

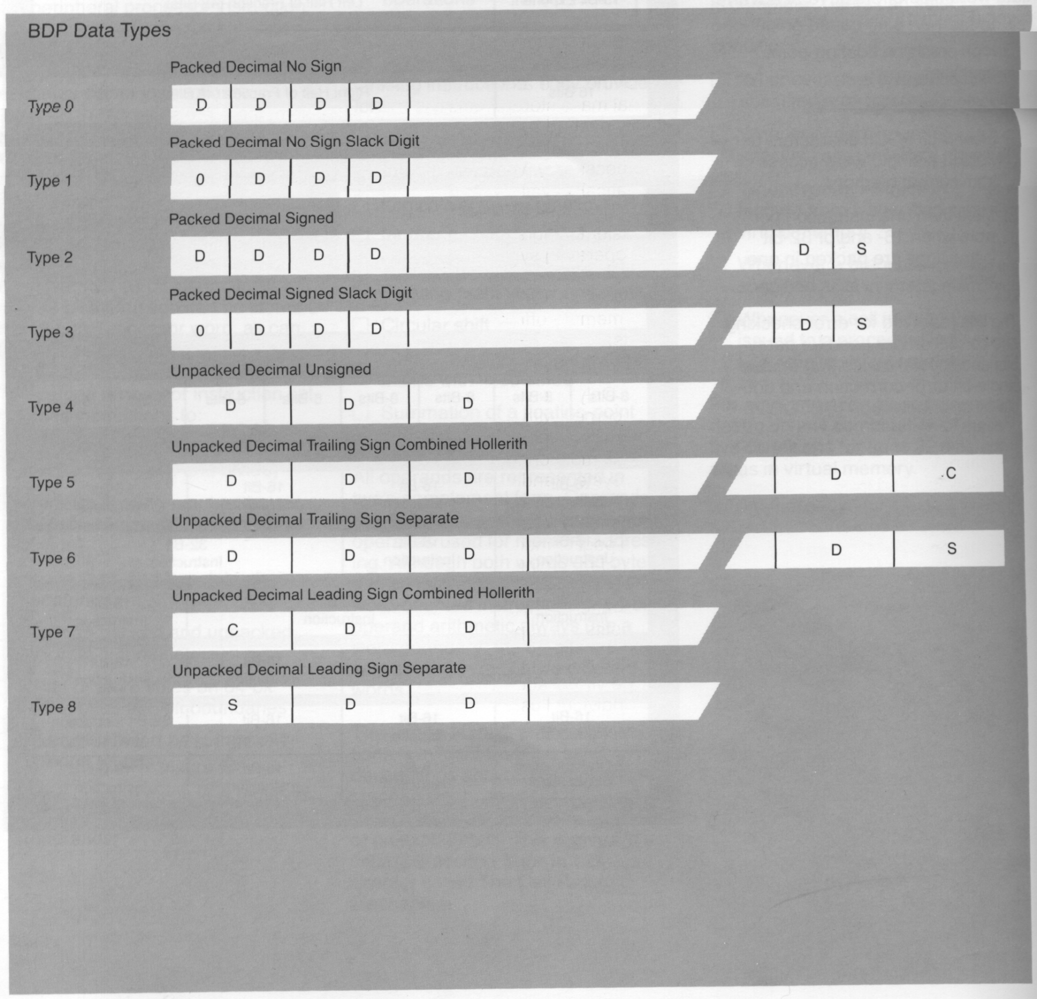
The bits reserved for error checking and correction are used for parity bits and special codes that enable single-bit error correction and double-bit error detection (SECDED).

Figure 18


Single Precision Floating-Point Quantities
15-Bit Exponent
48-Bit Fraction
Sign
Double Precision Floating-Point Quantities
15-Bit Exponent
Left Half of Fraction (48 Bits)
Sign
16 Bits
Right Half of Fraction (48 Bits)
Not used as input, but set on output to the exponent and the sign of the left half of the fraction
32-Bit, Signed, 2's Complement Integers (for Memory Addressing)
32-Bit Integer
64-Bit, Signed, 2's Complement Integers
64-Bit Integer
Eight 8-Bit ASCII Characters
8-Bits
8-Bits
8-Bits
8-Bits
8-Bits
8-Bits
8-Bits
8-Bits
Multiple Instructions
16-Bit Instruction
32-Bit Instruction
32-Bit Instruction (Left Half)
32-Bit Instruction (Right Half)

In addition, decimal arithmetic data is available in 180 state. (Figure 19.)

Figure 19


BDP Data Types
Type 0
Packed Decimal No Sign
D D D D
Type 1
Packed Decimal No Sign Slack Digit
0 D D D
Type 2
Packed Decimal Signed
D D D D
D S
Type 3
Packed Decimal Signed Slack Digit
0 D D D
D S
Type 4
Unpacked Decimal Unsigned
D D D
Type 5
Unpacked Decimal Trailing Sign Combined Hollerith
D D D
D .C
Type 6
Unpacked Decimal Trailing Sign Separate
D D D
D S
Type 7
Unpacked Decimal Leading Sign Combined Hollerith
C D D
Type 8
Unpacked Decimal Leading Sign Separate
S D D

Real Memory in 180 State

The CYBER 180 series software and hardware architecture recognize a real memory address of 31 bits which is an addressable real memory of about 268 million 64-bit words. Although such memories are dependent on new circuit technology, the introduction of this technology will not require changes to the system architecture.

In the CYBER 180 series, paging (real memory management) and virtual memory (user memory management) are isolated. A page is a unit of real memory. To allow performance tuning and to accommodate the larger memories of the future, the paging scheme allows different size pages to be used. Page size can range from 2,048 bytes to 65,536 bytes in increments of powers of two; software currently supports 2,048 bytes to 16,384 bytes. Page size is set at system initialization and is then used by all software. It can be changed from the console when the machine is idled down without any effect on software, user or system. Variable page size is effectively isolated from the user. The ability to select page sizes is both a tuning tool and a guarantee that, as memories grow, there is no need to redesign the system.

Virtual Addressing and Program Isolation in Memory

The 180 state uses a virtual memory mechanism to address and isolate programs in memory. The obvious advantage of the traditional virtual memory mechanism is, of course, to allow a user to have a very large address space without being constrained by real memory. The program can far exceed the real memory of the machine because most of the code and data resides in external mass storage until it is actually needed by the program; overlays and other such techniques are unnecessary. Virtual memory is automatically available to the user. The program is easier to develop and easier to migrate between different operating systems.

In addition, the CYBER 180 series virtual memory offers some unique benefits:

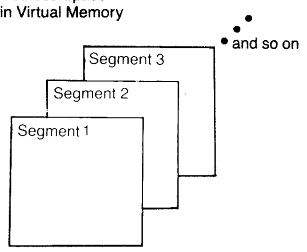
- ☐ NOS/VE itself uses virtual memory. Only one system process is concerned with managing real memory (paging); other than that single monitoring routine, both the system and user programs are totally separated from the real memory paging mechanism. The paging mechanism and the rest of monitor run in real memory but address it using the virtual mechanisms so that security benefits still apply.
- ☐ These machines offer extremely large virtual address spaces. For an individual user, a program can have a virtual address space of 8.8 x 10^{12} bytes.
- ☐ Code is automatically and completely shared (that is, reentrant). Data can also be shared but, for security reasons, it can be done only by explicit action by the programmer.
- ☐ The virtual memory mechanism is also used as a security mechanism. Units of virtual memory called segments have certain attributes that determine what operations can be performed in that segment and who can perform the operation.

As in 170 state, each user's program has an address space. The purpose of the address space is the same and the method of switching programs is the same. However, using the 180 state virtual memory mechanism, there are several different aspects about the address space itself.

The program's address space in virtual memory is made up of segments. (Figure 20.)

Figure 20

Address Space in Virtual Memory



When a segment is active, it contains zero, one or more pages of real memory. (Figure 21.)

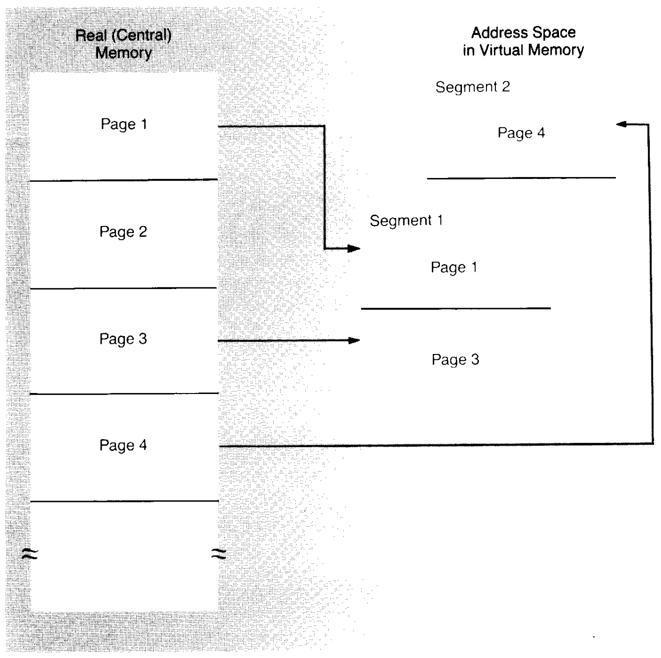
Unlike 170 state, when a program is read into real memory, it needn't be in contiguous positions. It can be loaded anywhere there is room in memory subject to the total number of pages available.

Other than to show the connection here between real memory and virtual memory, pages have no significance to the user, or to most of the system. The size of a page (that is, the number of central memory words contained in it) can be changed at system initialization with absolutely no effect on user programs or system software.

The address space itself can have from 1 to 4096 segments with up to 2,000 million bytes (262 million words) per segment. One purpose of segments is actually to separate, or segment, programs into sections of executable code (the instructions) and data. This has two advantages: it provides for built-in security features, and it allows parts of programs (either code or data) to be shared.

First, we'll discuss the security aspects. The first level of security is the address space of the program as described earlier. Second are access attributes. Third are rings.

Figure 21



Every segment has certain access attributes associated with it: read, write, execute, or a combination of these three. Say for example that segment 31 (Figure 22) contains executable code; it has the execute attribute associated with it. Segment 32 is a read/write segment that holds data that can be modified. Segment 33 contains constants and therefore is defined as a read-only segment.

Figure 22

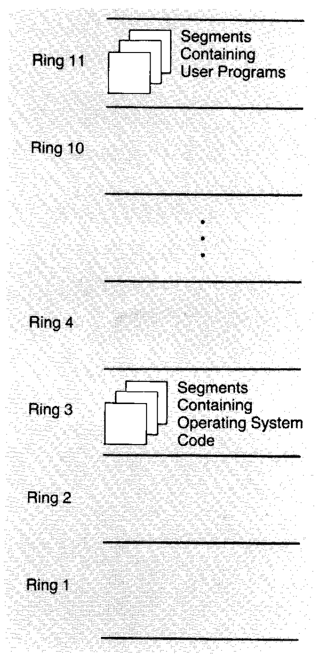
Segment 31 (Execute)	Segment 32 (Read/Write)	Segment 33 (Read-only)
Code	Variable Data	Constant Data

Because of the access attributes associated with it, each segment is protected from inappropriate use. For instance, nothing can be written into an execute or read-only segment so the information already contained there is safe. Typically, a program has at least a code segment, a data segment, and a binding segment (the last of which is used for procedure calls and is discussed later in this chapter under The Call/Return Mechanism).

The access attributes determine *what* can be done in a segment. In addition, each segment has another property that determines *who* can use the segment. This property is called a ring number. You can think of the system being within 15 rings, or levels, where each ring, or level, is more or less privileged than those adjacent to it.

In 180 state, the lower the ring number, the more privileged, and thus more secure, the information. Most operating system code executes in the lower (inner) rings with less secure user programs executing in the higher (outer) rings. (Figure 23.) For example, ring 11 accepts calls from ring 14 but only at designated entry points called gates. A more extensive example of ring numbers is given later in this chapter under Using Virtual Memory as a Security Mechanism.

Figure 23


Ring 11
Segments Containing User Programs
Ring 10
Ring 4
Ring 3
Segments Containing Operating System Code
Ring 2
Ring 1

Rings allow code that has different levels of protection, for example operating system routines and user programs, to execute in the same address space using the same conventions and rules for procedure calling and parameter passing as code in an environment with single-level protection. In addition, the 15 levels of rings allow installations and protected applications to isolate their code from end users while protecting the operating system from all of them.

The second advantage of segmenting programs is the ability to share parts of the program without affecting others. This allows users with different sets of data to access the same code. This sharing is done automatically; users needn't do any special programming to take advantage of it. The way segments are actually shared is discussed in more detail later in the next section where the actual address translation is explained.

How Virtual Addresses Are Translated to Real Memory Addresses

Within a program's address space, only virtual addresses are used; the program sees only those virtual addresses local to its address space. This virtual address known within the address space is called the process virtual address (or PVA). Because it is unique to the address space, this address is translated into a virtual address that is known system-wide; it is called the system virtual address (or SVA). Finally, the system virtual address is translated to a real memory address (or RMA). (Figure 24.)

In the context of NOS/VE, a program consists of one or more tasks where a task is the smallest executable unit of code. These tasks could be a combination of user tasks and system tasks that form what is typically called a program.

The process virtual address in the task address space contains the number of the segment in which it is contained and a byte number that is the offset to the requested memory byte. (Figure 25.)

Figure 24

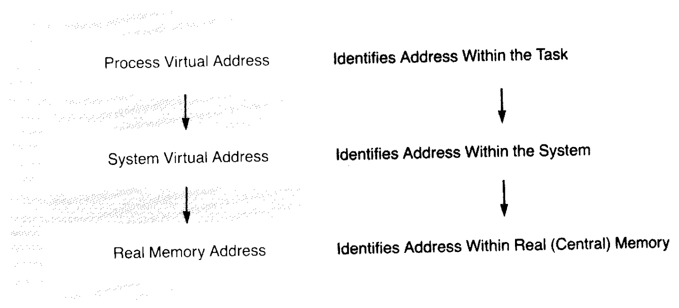
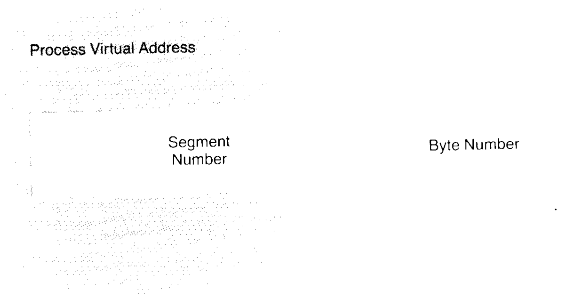
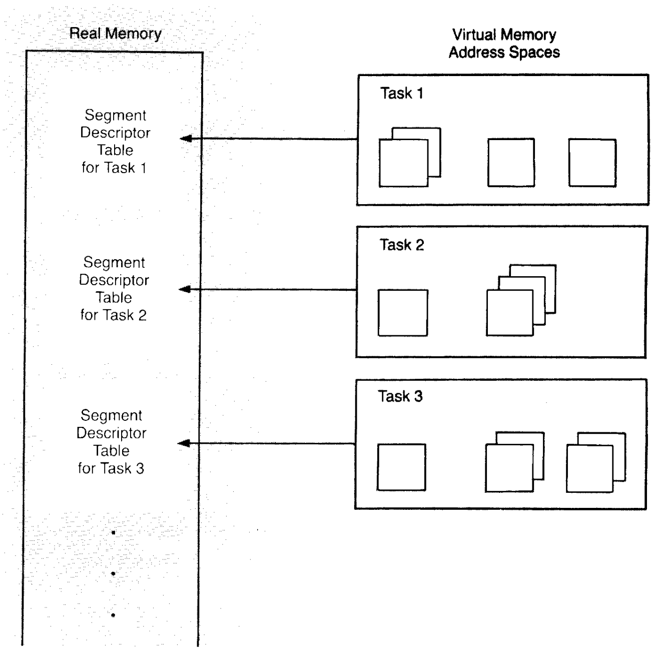

Process Virtual Address
Identifies Address Within the Task
System Virtual Address
Identifies Address Within the System
Real Memory Address
Identifies Address Within Real (Central) Memory

Figure 25



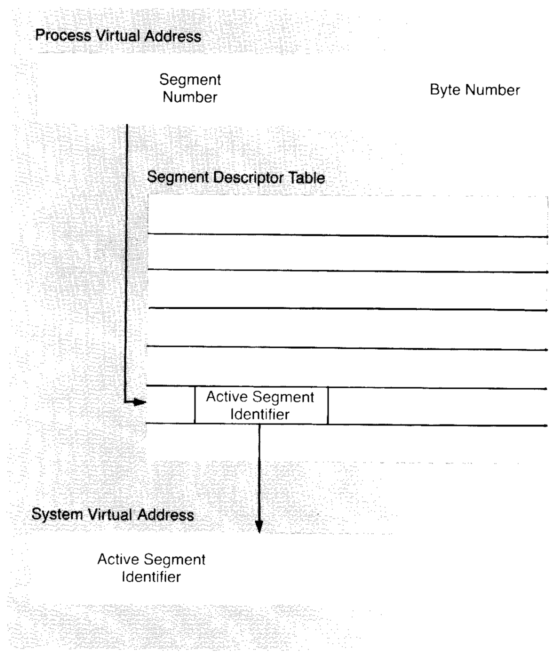
The segment number is used as an index into a table called the segment descriptor table. There is a segment descriptor table for every task in the system. (Every task has its own virtual address space.) (Figure 26.)

Figure 26


Real Memory
Virtual Memory Address Spaces
Segment Descriptor Table for Task 1
Segment Descriptor Table for Task 2
Segment Descriptor Table for Task 3
Task 1
Task 2
Task 3

The entries in the segment descriptor table contain the information needed to continue the translation process. One of the fields contained in each table entry is called an active segment identifier. This field identifies each of the segments that are active in the system and it replaces the process segment number to form part of the system virtual address. (Figure 27.) The active segment identifier is defined by the system and varies from one execution of a program to the next.
The process segment number is assigned at load time and is unique to that task.

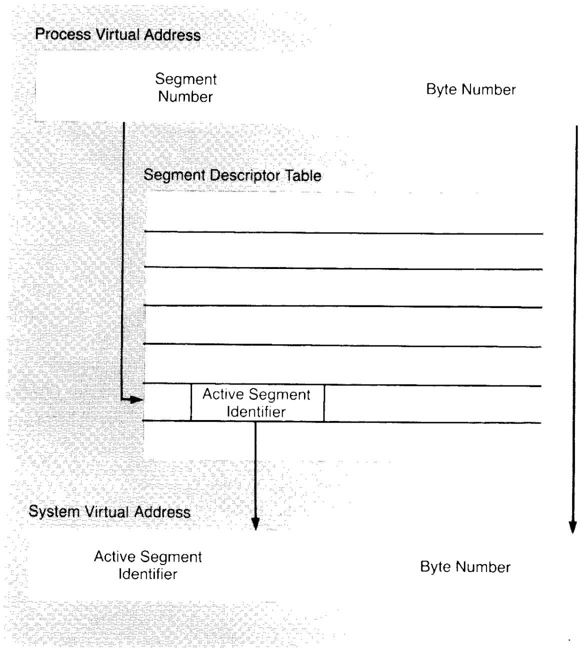
Figure 27



The shading shown in these figures reflects bits that are not discussed; it does not mean the bits are unused.

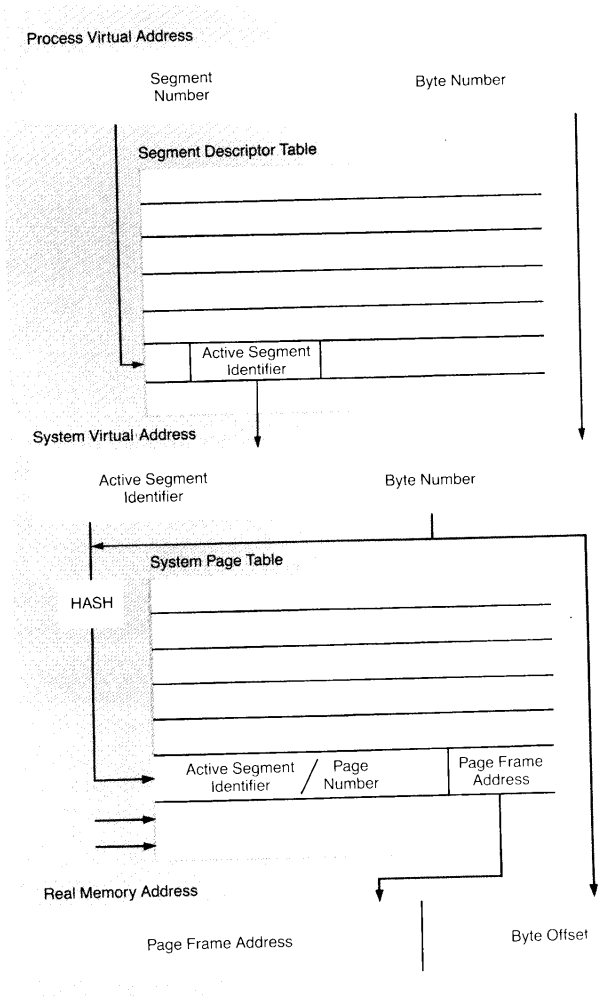
To form the remainder of the system virtual address, the byte number of the original process virtual address is copied. (Figure 28.)

Figure 28



The system virtual address then goes through a hashing algorithm that points to the starting point of another table in central memory called the system page table. Consecutive entries in the system page table are searched until the entry describing the desired page in real memory is found or until a page table entry that causes the search to stop is encountered. (Figure 29.)

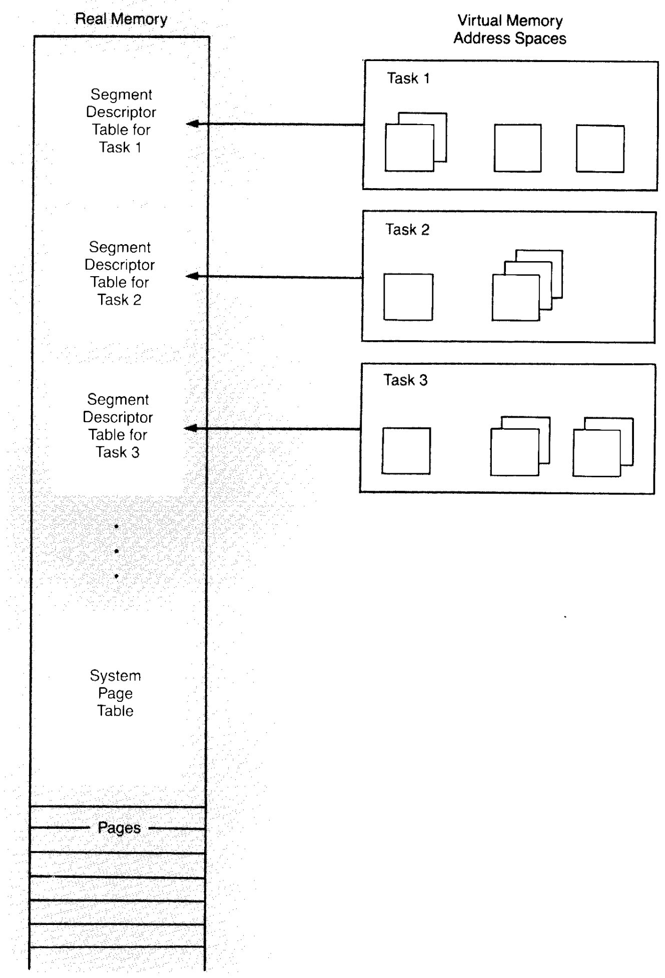
Figure 29



There is only one system page table in a mainframe regardless of the number of central processors. (Figure 30.)

If the requested page is not in the system page table, the central processor suspends execution of the task and the operating system either reads the requested page from mass storage (if the page already exists) or assigns an available page of memory (if the page is being created). The operating system then adds a new entry in the system page table that describes the page and resumes execution of the interrupted task.

Figure 30

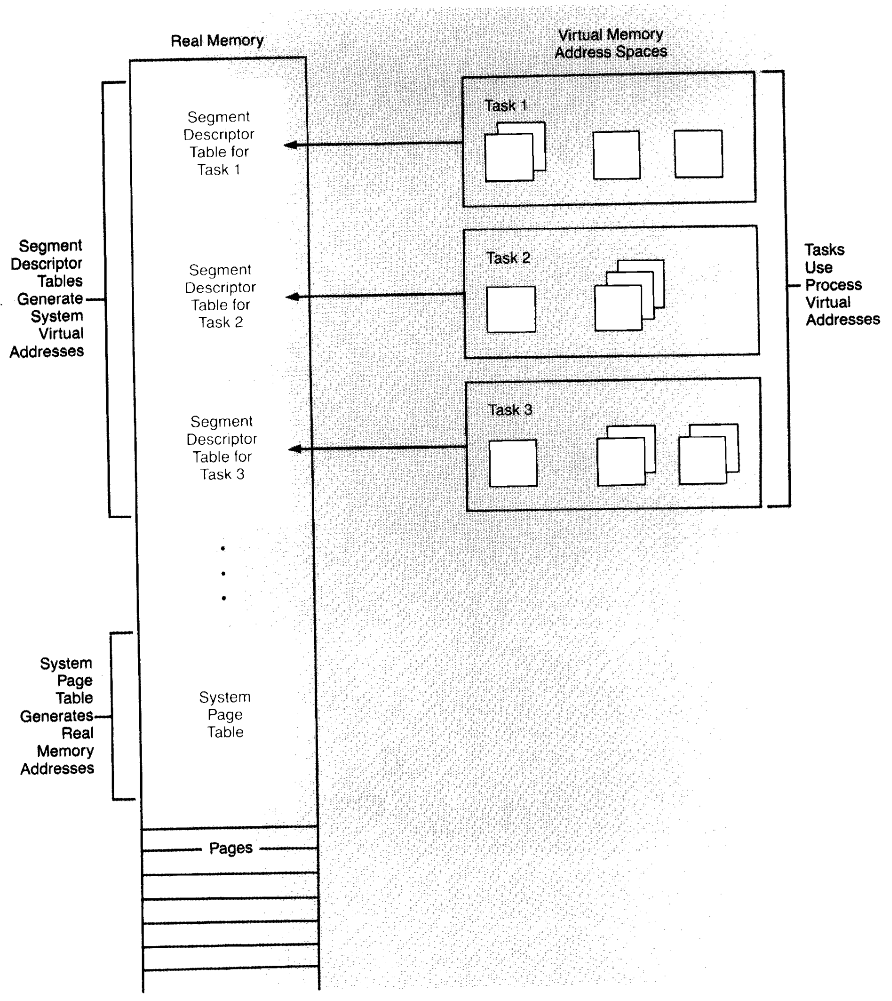


Real Memory
Virtual Memory Address Spaces
Segment Descriptor Table for Task 1
Segment Descriptor Table for Task 2
Segment Descriptor Table for Task 3
System Page Table
Pages
Task 1
Task 2
Task 3

In summary, virtual memory address spaces use process virtual addresses. The segment descriptor tables generate system virtual addresses. The system page table generates real memory addresses. (Figure 31.)

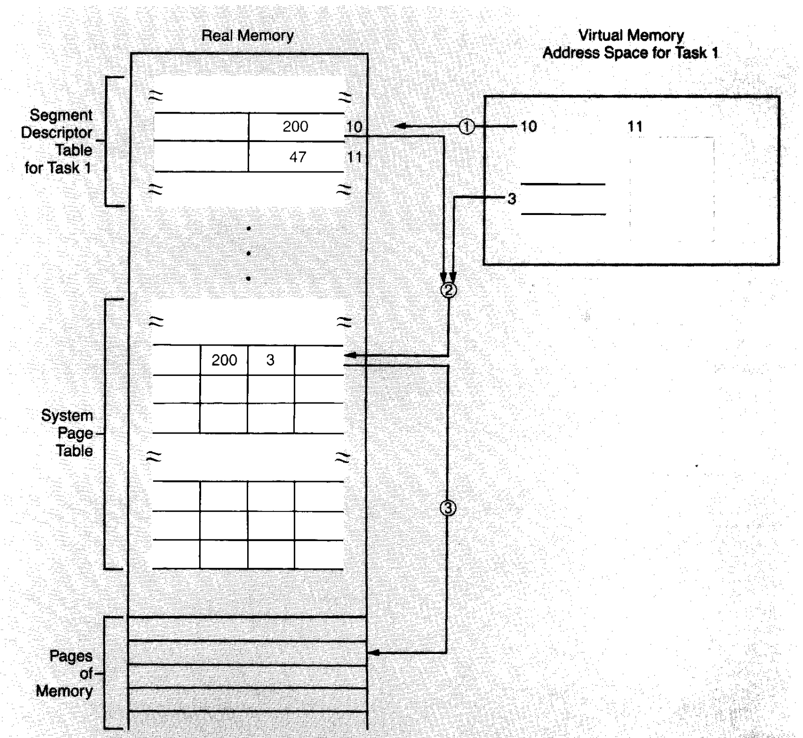
Figure 32 illustrates a very simple example of the process. It shows a task address space and associated segment descriptor table. When a reference is made to an address in the task, that address is the process

Figure 31


Real Memory
Virtual Memory Address Spaces
Segment Descriptor Table for Task 1
Segment Descriptor Table for Task 2
Segment Descriptor Table for Task 3
System Page Table
Pages
Task 1
Task 2
Task 3
Segment Descriptor Tables Generate System Virtual Addresses
System Page Table Generates Real Memory Addresses
Tasks Use Process Virtual Addresses

virtual address unique to the address space. It is translated via the segment descriptor table to a system virtual address. In this example, 10 is the segment number; it is used as an index into the segment descriptor table to find the active segment identifier of 200. This number and the byte number from the process virtual address (in this case, 3) are hashed to a pointer into the system page table. This table is searched to find the entry that describes the desired page in real memory. The contents of this real memory page are then mapped as part of segment 10 within the program.

Figure 32



Real Memory
Virtual Memory Address Space for Task 1
Segment Descriptor Table for Task 1
200
10
47
11
10
11
3
System Page Table
200
3
Pages of Memory

① The segment number is an index into the segment descriptor table.

② The active segment identifier and byte number form the system virtual address which is hashed to point into the system page table.

③ An entry in the system page table describes the real memory address.

With the address translation explained, the method for sharing code and data becomes quite easy to understand. It is possible both because of the separation of code and data segments in the task, and the fact that two unique virtual addresses can point to the same real memory address. The two tasks in Figure 33 share a single code segment. The unique process virtual addresses in each task that reference the shared segment are translated to the same system virtual address. The system page table then produces the single real memory address. Thus, the same page appears to be present in two address spaces, one in segment 12 and one in segment 37. The two tasks share the same area of memory.

Conceptually, this address translation occurs every time the central processor translates a virtual address. In actuality, however, all models of the CYBER 180 series have buffer memories that hold the most recently translated segments and pages.

Managing the System Page Table

The hardware has been designed with dynamic paging in mind. The algorithm for determining when pages are brought into and taken out of real memory is the responsibility of the operating system. However, two flags kept in the system page table entries help in this process. Whenever a page is used (read, written, or executed), the hardware sets bit 2 in the page table entry. Whenever a page is modified (written), the hardware sets bit 3 in the page table entry. These bits then have the following meanings:

- 00 (both bits clear) means the page is unused and unmodified.
- 01 (bit 3 set) means the page is unused but modified. This status can arise from software algorithms.
- 10 (bit 2 set) means the page is used but not modified (read).
- 11 (both bits set) means the page is used and modified (written).

Pages are chosen as candidates for reassignment based on the value of these bits and their least recently used status. These bits are never cleared by hardware. They are cleared by the software process that manages the system page table. The modified bit is cleared when the page is written to disk. The used bit is cleared periodically to determine when a page is no longer being used. Pages are reassigned on a least recently used basis.

The Call/Return Mechanism

The call/return mechanism is the technique used in 180 state for transferring control between procedures (subroutines). It is also used to assist in code sharing and transferring across ring boundaries. It is designed to satisfy the requirements of block structured languages, in particular CYBIL, that permit recursive calls.

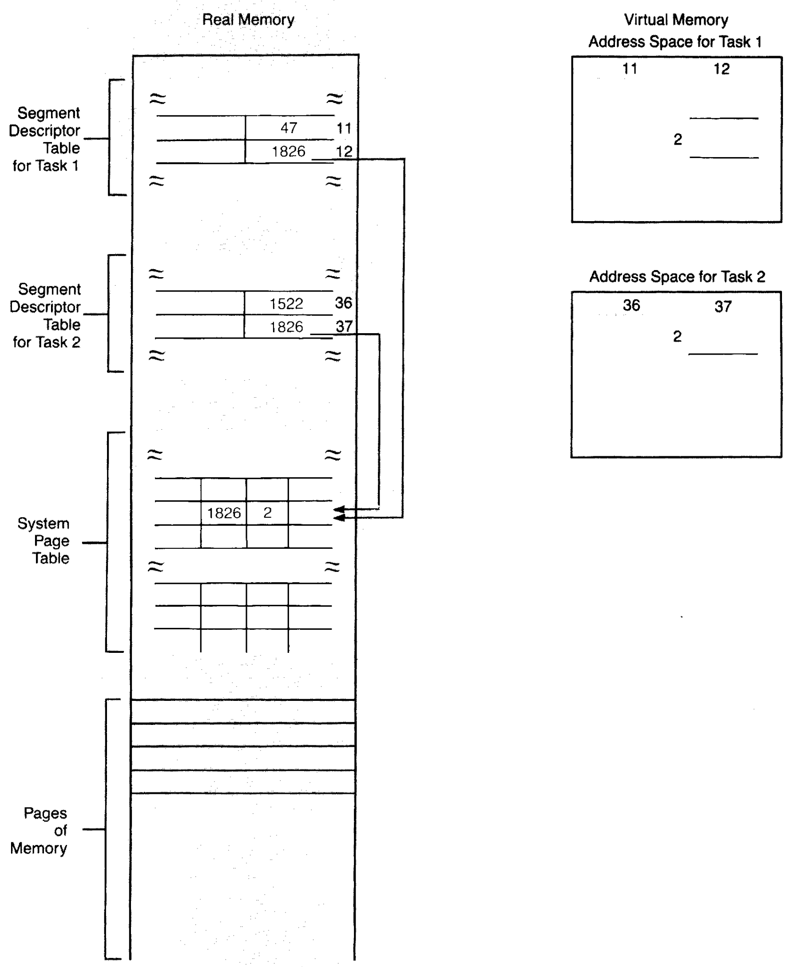
Procedures (or subroutines) in a block structured language are organized into a series of nested blocks.

Variables can be allocated at a number of times during program execution. Memory for static variables is allocated at the beginning of the program and remains until the end of the program. Dynamic variables are allocated each time a procedure is called and disappear when the procedure returns to the caller. This allocation occurs in a stack.

A stack is an area in memory that expands and contracts according to the procedure call logic of the program. The area in the stack associated with a call is known as a stack frame. For any program, there is one stack segment for each of the 15 rings being used by the task.

Each time a procedure is called, a new stack frame for that procedure is created. The stack is managed primarily by the call/return hardware mechanism. The main objective is to allow each instance of a call to a procedure to be separate. This allows, among other things, recursive procedures and support for one of the two levels of interrupt, the trap.

Figure 33



Real Memory
Virtual Memory
Address Space for Task 1
11
12
2
Address Space for Task 2
36
37
2
Segment Descriptor Table for Task 1
47
11
1826
12
Segment Descriptor Table for Task 2
1522
36
1826
37
System Page Table
1826
2
Pages of Memory

Figure 34 shows two sets of nested blocks in program A: B and C, and D and E. Procedure C uses variables defined in A and in the procedures B and C. The actual location of these variables is maintained by a static link which is held in the stack frame for each procedure. This linkage is called static because its length is known by the compiler at compilation time and never changes.

Figure 34

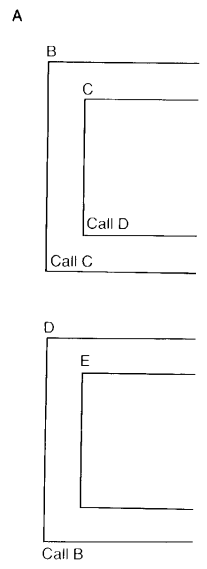
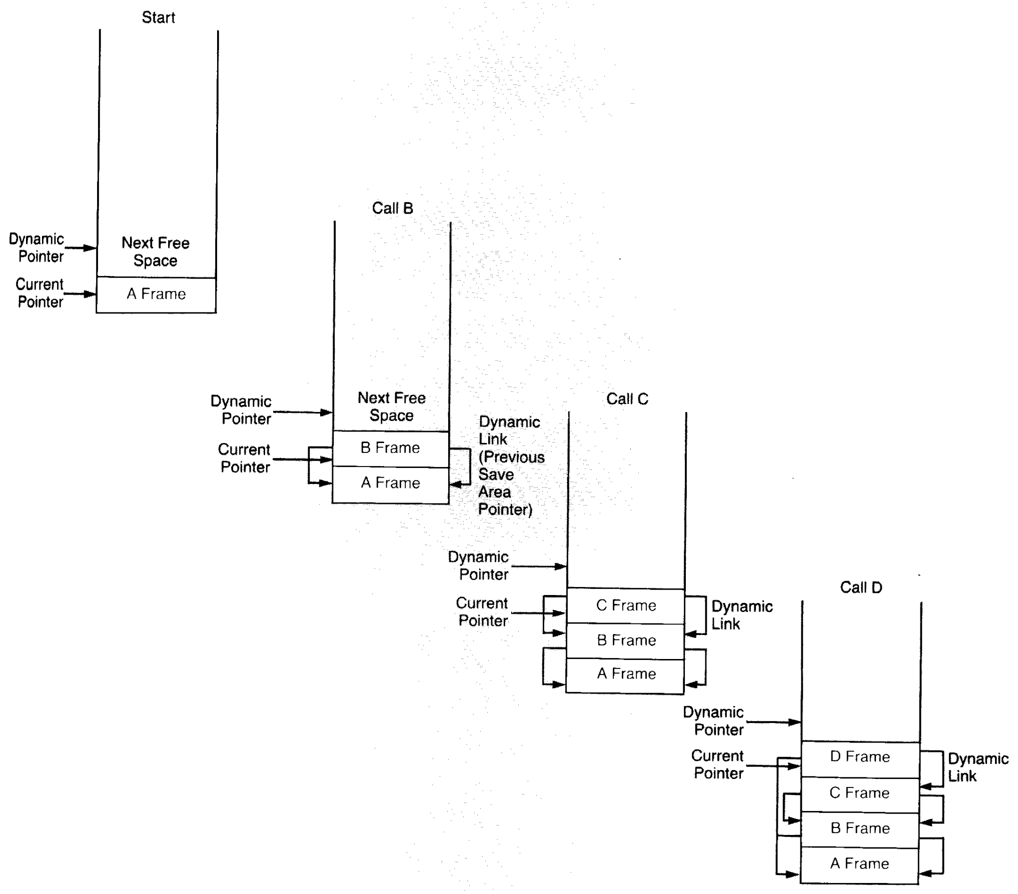


Figure 35 shows the stack mechanism. The process starts by creating a stack frame for the dynamic variables in the module A. A current stack frame pointer points to the beginning of the stack frame and a dynamic space pointer points to the next available space in the stack. (The dynamic space pointer is established by software.) When procedure B is called, procedure A's environment is saved and a stack frame is created for procedure B. A dynamic link is created pointing to procedure A's stack frame and, in this case, a static link pointing to the same stack frame. The dynamic link is called the previous save area pointer and it is automatically updated by a call and return operation by the hardware. The dynamic link is called dynamic because its length changes during program execution.

A call to procedure C follows in much the same way. Again the static and dynamic links simply point to the previous stack frame. However, when procedure C calls procedure D, the dynamic link points to the previous stack frame but the static link points to the stack frame for module A. It does not point to those declared in blocks B and C which are contained within A but do not contain D. The reason for this is that procedure D is a block within the base module A; thus, procedure D has access to variables declared in module A but not to those declared in blocks B or C.

Figure 35

Stack Frames



On each procedure call, the next stack pointer is updated to point to the next available space within the stack. This is a software function.

The frame contains the saved environment, mandatory registers, and other registers (Figure 36) where the saved environment exists only after another procedure has been called, not for the current procedure. For example, initially the stack is as shown in Figure 37.

After a call is issued, the stack contains two frames as shown in Figure 38.

Figure 36

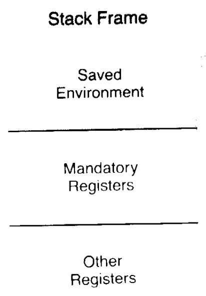


Figure 37

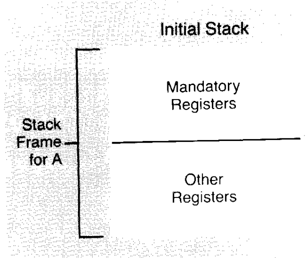
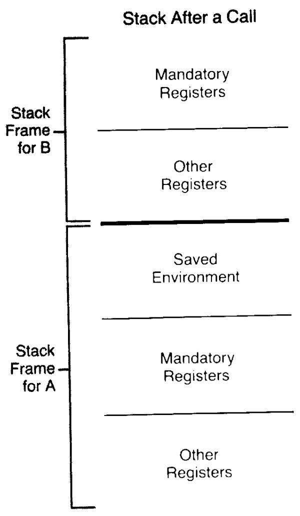


Figure 38



Because each time a procedure is called the caller's environment is saved, a procedure can be reentered or called recursively. However, this is true providing all code is organized into pure procedures. No code modification is permitted.

The call/return mechanism provides facilities for protection and dynamic linking. First, it is necessary to understand the hardware support for the basic mechanism.

The Call Mechanism

A stack segment is created by the operating system for each ring of execution. A top of stack pointer for each of these stacks is kept in the exchange package. Its main use is to show which segment contains the stack on a transfer between rings. Whenever a procedure calls another procedure, the caller's environment is saved in the stack frame save area. The first four words of this area are stored unconditionally; the remaining words are stored under the control of the caller.

The caller specifies a stack frame by defining which X and A registers are to be saved in addition to those saved by default. When the callee returns to the caller, these registers are automatically restored. Thus, the operation of the hardware supports a software calling convention whereby the caller saves the environment.

The following basic steps are followed for a call* (with the pointers being updated accordingly):

1. The caller's environment is saved.
2. The caller's stack frame is pushed.
3. The P register is updated to point to the first instruction of the callee to be executed.

There is a single return instruction which simply inverts this process:

1. The callee's stack frame is popped.
2. The caller's environment is restored.
3. The P register is updated (from the caller's environment) so that it points to the first instruction following the original call to be executed.

The caller's environment is saved in the caller's stack. In fact, it is saved at the top of the caller's stack. The callee's stack frame is not created automatically. The current stack frame pointer is updated to point to the first entry in the stack frame, but it is the responsibility of the callee to reserve the appropriate amount of space in the stack.

*180 state supports two forms of the call instruction that may be regarded as general purpose (CALL INDIRECT) and special purpose (CALL RELATIVE) calls. The CALL INDIRECT instruction may call into a different segment in a different ring. The CALL RELATIVE instruction calls only within the same segment. Although the same basic mechanism applies to both forms of the call, the general purpose version must guarantee the privacy of the callee and the caller who may have quite different privileges.

The Return Mechanism

The basic return mechanism pops the callee's stack frame and restores the caller's stack frame as the active frame. In other words, the environment which exists following the execution of a return instruction is exactly that which existed prior to the execution of the associated call instruction. Figure 39 shows the changes that occur in the stack when this sequence of instructions is followed:

1. Intra-ring call
2. Inter-ring call from ring 11 to ring 3
3. Return
4. Return

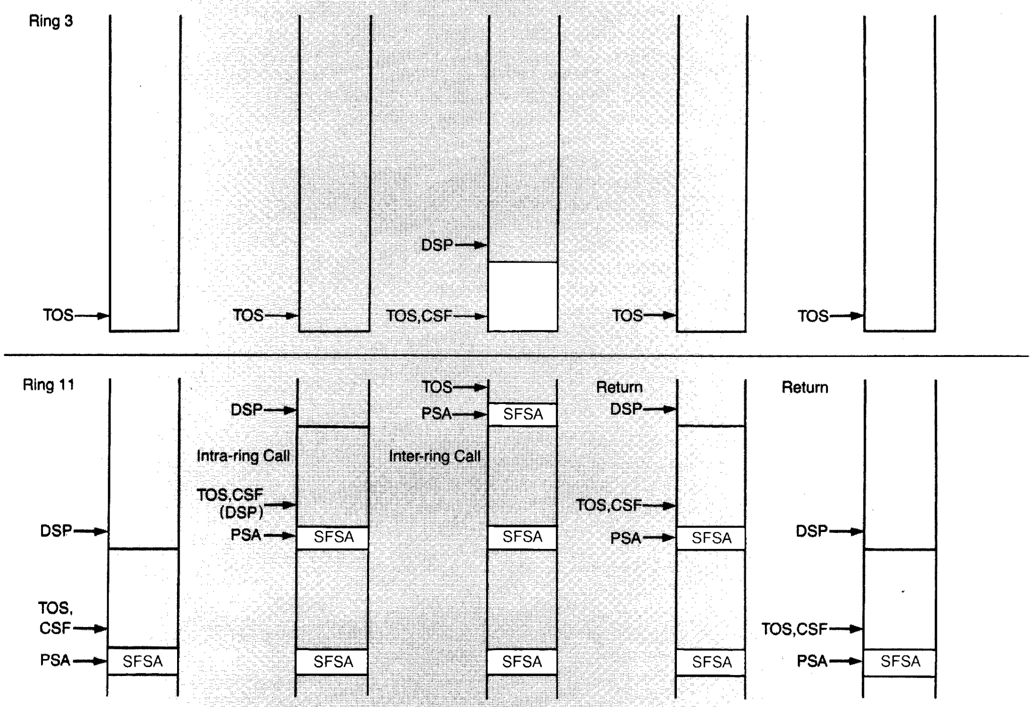
Calls are typically within rings. Calls can be made to more privileged rings which can access less privileged data and write data back in a place where the less privileged code can access it. The inverse is not true. Calls from more privileged procedures to less privileged procedures are not permitted. Such returns would be uncontrolled and, therefore, are not allowed.

The callee's ring number may appear in any A register used by the callee, not just those saved by the caller. For security reasons, the ring number must not be returned to the caller, so a check is made by the return instruction to ensure that no A register is returned to the caller with a ring number more privileged than the caller's ring number.

The caller's privileges, maintained in the P register, are automatically restored when the caller's P register is loaded from the stack frame save area.

Typically, processes start execution in their outermost ring. Stacks in all rings will be empty except for the one in the primary ring of execution. As calls are made inward, entries are made in other stacks which will be emptied as return instructions are issued. There are 15 stacks because of the security of the system. Because the stack holds the dynamic variables for an executing process, that process has read/write access to the stack. If there were only a single stack, then an executing process could make a call to a procedure in an inner ring and then access that procedure's dynamic variables which would be at the top of the stack. The only way to prevent this would be for the callee to zero out all dynamic variables used. This would be prohibitively time-consuming.

Figure 39



Key:

CSF	—Current Pointer
DSP	—Dynamic Pointer
PSA	—Previous Save Area Pointer (Dynamic Link)
SFSA	—Stack Frame Save Area
TOS	—Top of Stack Pointer

The Pop Mechanism

There are times, typically in the presence of an error or a nonlocal GOTO, when it is necessary to eliminate an entry or a number of entries from a particular stack. Since these entries will have been created by a series of calls, a similar series of returns will accomplish the required purge. However, when the purging is to be completed without executing intervening instructions, this can only be achieved by an appropriate software sequence or by issuing the pop instruction that has been provided for this purpose. The pop instruction simply moves the current stack frame pointer, previous save area pointer, and top of stack pointer eliminating the stack frame but not changing the P counter. Figure 40 shows an example where calls have been made three deep into the structure of a program and then the entire set of calls aborted.

Pop instructions can only be issued within the current ring of execution. Access violations are not checked, and if a pop instruction is attempted across rings, then the instruction execution is inhibited and the program interrupted. (The actual code sequence that would be generated would be a loop because the number of frames to pop is not known at compilation time.)

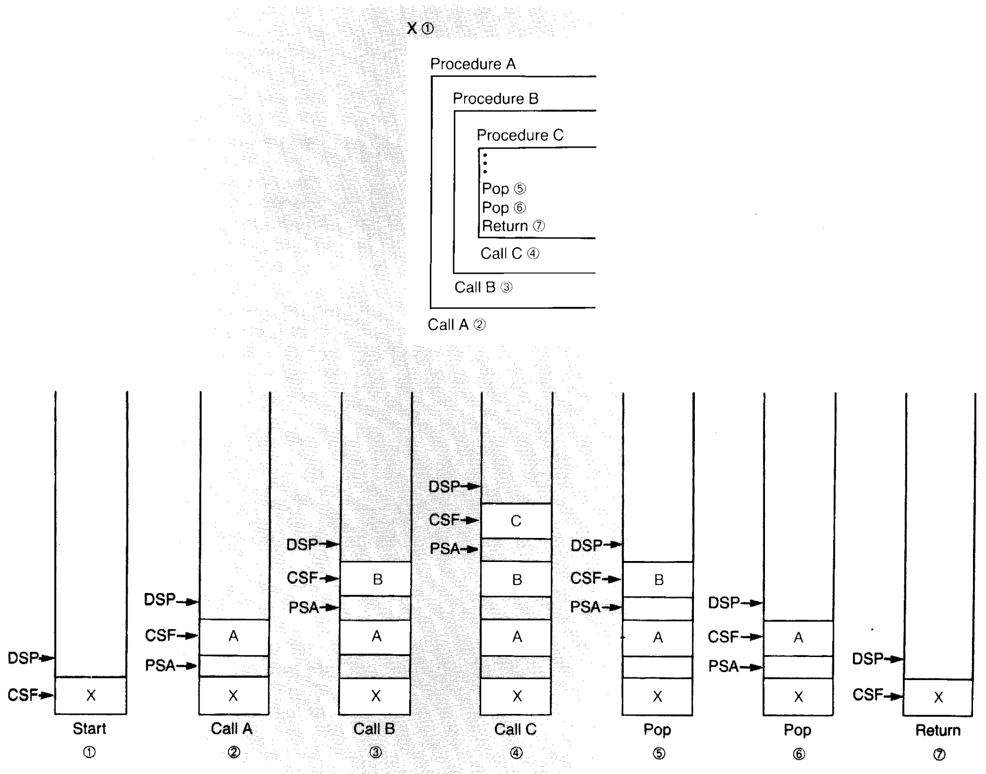
The Binding Section and Code Sharing

An implication of protecting code is that the protected code can be entered only where it expects to be entered. It is important that the entry to procedures be carefully controlled. That is, procedures must receive control only at those points where they expect to receive control, their entry points. These special entry points are called gates. To make this controlled entry possible, procedures are not entered directly* but are entered via a pointer to the procedure. This pointer is held in a binding section (in other words, a binding segment). All such pointers are placed in the binding section by the loader and the call instruction then guarantees that the call is made via a binding section.

180 state uses one copy of a code segment that is shared by several users (this is called re-entrancy). For example, the FORTRAN compiler exists in only one place in real memory, but each user has asked the compiler to operate on a different compilation unit. There must be nothing in the code segment that makes a direct reference to data which is modified. This is accomplished by placing pointers to such data in a binding section that is created along with each code module, and then give the address of the binding section to the callee when the procedure is called.

*This is true only for calls between segments. Relative calls can enter anywhere within the procedure; that is, within the segment.

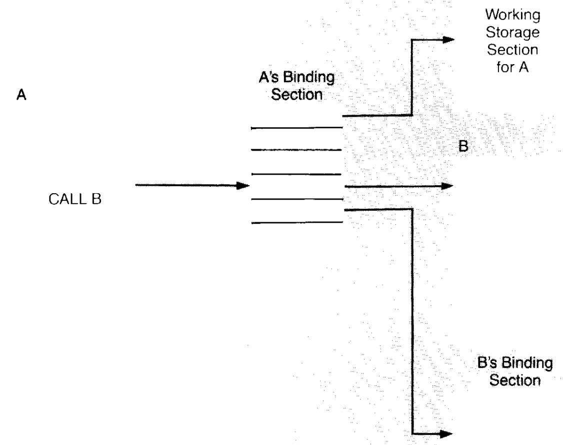
Figure 40



Key:

CSF —Current Pointer
DSP —Dynamic Pointer
PSA —Previous Save Area Pointer (Dynamic Link)

Figure 41



Each task has code (which it may be sharing with other tasks) and data which is typically unique to itself. The binding section is in a separate segment and is identified uniquely by its segment descriptor entry. It contains pointers to external procedures and pointers to static variables (that is, variables that are allocated at compilation time).

When a procedure calls another procedure that is defined externally, the call points into the binding section. (Figure 41.) A pointer points to the first executable statement in procedure B. A flag indicates that the procedure being called is an external procedure and, therefore, the next entry in the binding section is the pointer to the callee's binding section.

Using Virtual Memory as a Security Mechanism

Even though code and data can be shared in a virtual memory system, security features of the CYBER 180 series constrain each task to an address space and prevent it from reading, writing, or executing code or data outside this address space, as it likewise prevents any other task or the operating system from doing the same thing in its address space. As mentioned earlier, the CYBER 180 concept of virtual memory allows the operating system to exist (as shared code) in each task's address space. The access protection mechanisms are therefore necessary to protect system code and user code/data.

Because every memory reference requires a translation from a virtual address to a real memory address, it is appropriate to check for security violations prior to accessing real memory. The access attributes and ring numbers described earlier are associated with every segment in a task. The operating system records the attributes and ring numbers for each segment in the segment descriptor table for each task (Figure 42) and the central processor references them when the process virtual address is being translated into a system virtual address. Thus the hardware can detect an attempt at an incorrect access and prevent it. You can view the relationship between address translation and security checking as shown in Figure 43.

Several tasks can actually share a segment with each task having different privileges. This is what enables efficient and safe code sharing and negates the necessity for every user to have a separate copy. When an address is referenced, the central processor compares the type of reference (that is, read, write, or execute) with the attributes in the segment descriptor table for that task. The request must match the attribute of the segment. For example, to execute a read instruction, that segment must have a read attribute. If a read instruction attempts to access a location in an execute-only segment, it is considered an access violation.

Figure 42

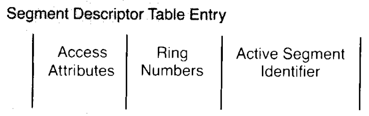
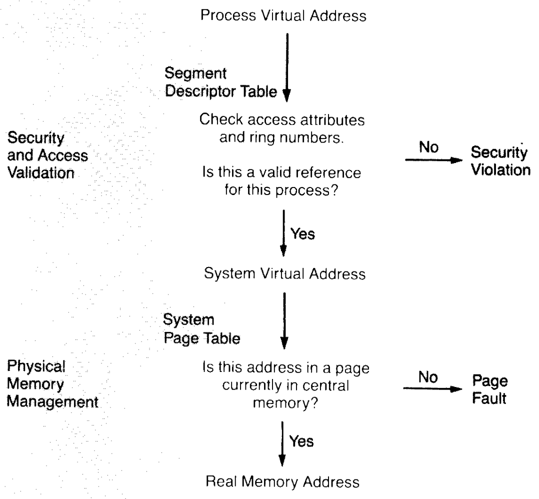


Figure 43



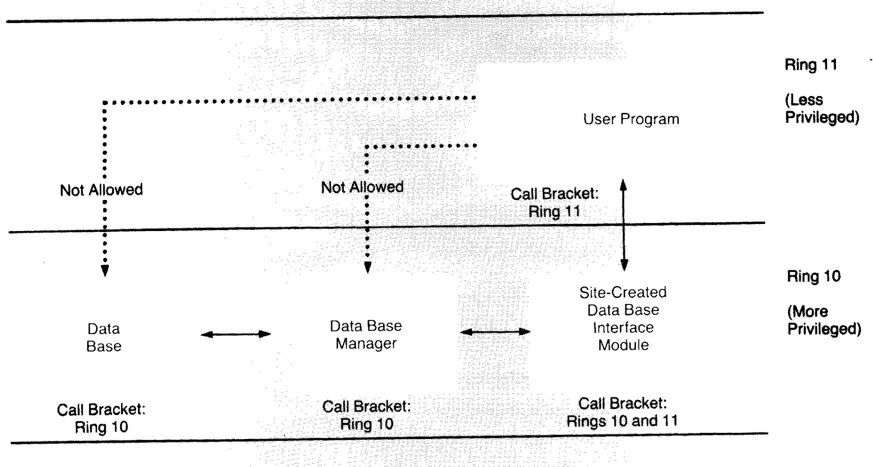
Process Virtual Address
Segment Descriptor Table
Check access attributes and ring numbers.
Security and Access Validation
Is this a valid reference for this process?
No
Security Violation
Yes
System Virtual Address
System Page Table
Physical Memory Management
Is this address in a page currently in central memory?
No
Page Fault
Yes
Real Memory Address

As mentioned earlier in this section, the system has 15 levels or rings of protection that also prevent unauthorized access. The lower the ring number, the higher the privilege given to code or data in that level. The operating system segments occupy the lowest rings with the least privileged user segments occupying the higher rings. The segment has associated with it ring numbers for certain operations (read, write, execute, and call). When a task references an address, the central processor compares the current ring number of the task attempting the read, write, execute, or call to the ring attributes of the segment where the address exists. If the referenced segment is in a ring that is accessible by the task at that point, it can be accessed. This allows codes with different levels of privilege to be in the same address space which provides for easy parameter passing and communication without losing security.

A segment always has four ring brackets (that is, ranges of ring numbers) associated with it. Code typically has an execute bracket that specifies where it can execute and a call bracket that specifies the ring from which it can be called. Data has a read bracket that indicates where it can be read and a write bracket that indicates where it can be written.

As an example, assume a site has a data base manager but doesn't want users to access it directly. (Figure 44.) Instead, all references must go through a data base interface module. The user program is in ring 11, which is less privileged than ring 10 where the data base modules (the manager and interface) are located. The interface module has a range of ring numbers such that it can communicate with rings 11 and 10 but the data base manager itself can only communicate with ring 10. The data base manager can be accessed only by the data base interface module and, thus, is isolated (or protected).

Figure 44



180 Peripheral Processors

NOS/VE uses peripheral processors for all input/output, thus achieving a tremendous overlap of functions and leaving the central processor available for user programs.

Peripheral processors on all models of the CYBER 180 series are functionally identical. The peripheral processor instruction set, registers, and channel usage are the same for both 170 and 180 state and are described more fully in the chapter on the system components.

In 180 state, peripheral processors share such features as:

- Memory and word size
- Method of reading and writing central memory words

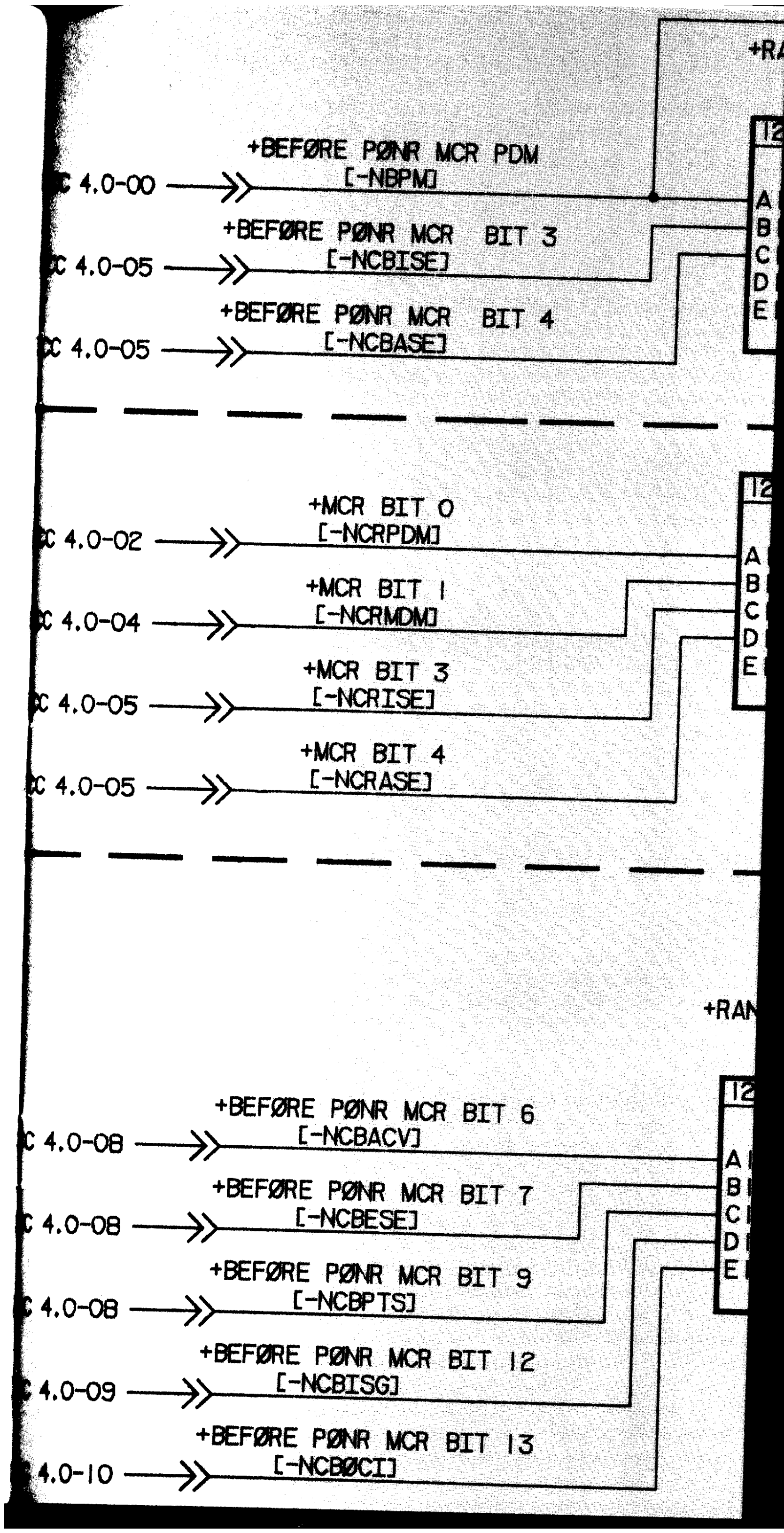
Every peripheral processor can communicate with any channel or other peripheral processor that is assigned to 180 state.

Memory and Word Size

In 180 state, a peripheral processor uses the full 16 bits as a memory word.

Reading and Writing Central Memory Words

In 180 state, 64-bit central memory words are read and written by peripheral processors using a hardware buffering mechanism that assembles and disassembles the words in 16-bit quantities. When a 64-bit central memory word is read, it is disassembled into four 16-bit words and sent to successive locations in the peripheral processor's memory. To write a central memory word, the hardware assembles four successive 16-bit peripheral processor words into one 64-bit word and sends it to central memory. The advantage of this process is that there is only one access to central memory rather than four.



170/180 State Interaction

NOS and NOS/VE Dual State

As mentioned earlier in this overview, dual state is the state of operation in which two operating systems, NOS and NOS/VE, are both present at the same time. NOS is executed in 170 state and NOS/VE is executed in 180 state. Actually the environments for each state are both present but the central processor executes in either one state or the other.

An important aspect of dual state is the dynamic sharing of the central processor. In the same way that programs are exchanged, 170 state and 180 state are exchanged. If the central processor is executing a program and would have to wait for some action to be taken before continuing, it switches programs and processes a different one. Likewise, if the central processor would have to wait for some action before continuing a program in one state and there is a program waiting in the other state, it switches states and works on the other program. The central processor never waits in one state if there is something of higher priority to do in the other state.

Dual state is made possible by the fact that the CYBER 180 series fully supports two instructions sets: one for 170 state and one for 180 state. (As was mentioned earlier, the peripheral processor instruction set is the same, but the central processor instruction set differs in each state.) When the central processor is executing in 170 state, it uses the 170 instruction set; in 180 state, it uses the 180 instruction set. The instruction set to be used is determined by a bit in the exchange package. Performance remains high in both states because neither is an emulation mode.

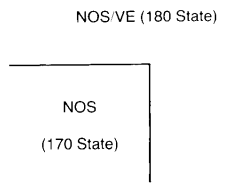
The 170 state environment is actually supported as if it were a special purpose 180 state program. Thus, parts of 180 state support 170 state but 180 state is transparent to any user programs executing in the 170 state environment.

The interface between the two environments, 170 state and 180 state, is the environment interface (or EI) which executes in privileged 180 state. The environment interface is the only part of the system that manages both virtual memory and real memory.

Although the hardware has been designed with dynamic paging in mind, it is not a prerequisite. When running in 170 state, static paging is used; that is, the entire 170 state environment is assigned to a single segment that cannot be paged. The NOS segment has read, write, and execute properties because NOS does not segment code and data from each other. 170 state operates in a virtual memory segment that has a size corresponding to the amount of memory defined to the system for 170 state. A real memory addressing mode exists within the hardware but it is only a pseudo mode since the hardware still goes through the address translation mechanism. However, there are no page faults.

In addition to real memory paging, the environment interface also handles security and error interrupts. Obviously it is itself a highly secure section of code which contains privileged instructions not available to other areas of the system.* NOS can actually be thought of as a program executing under the monitoring of the environment interface although the environment interface is obviously a part of NOS/VE. (Figure 45.)

Figure 45

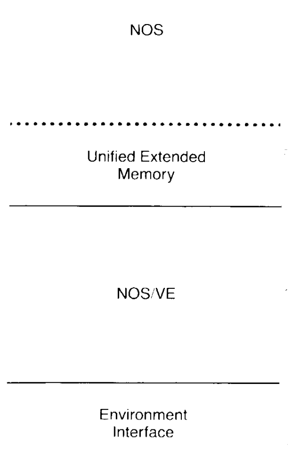


The same exchange principle used to switch programs is used to switch from 180 state to 170 state and vice versa. There is an interstate exchange package that contains all the information necessary to start a particular environment. First the interstate exchange package determines the state of operation (170 or 180 state), then the specific process exchange package determines the particular program to be executed.

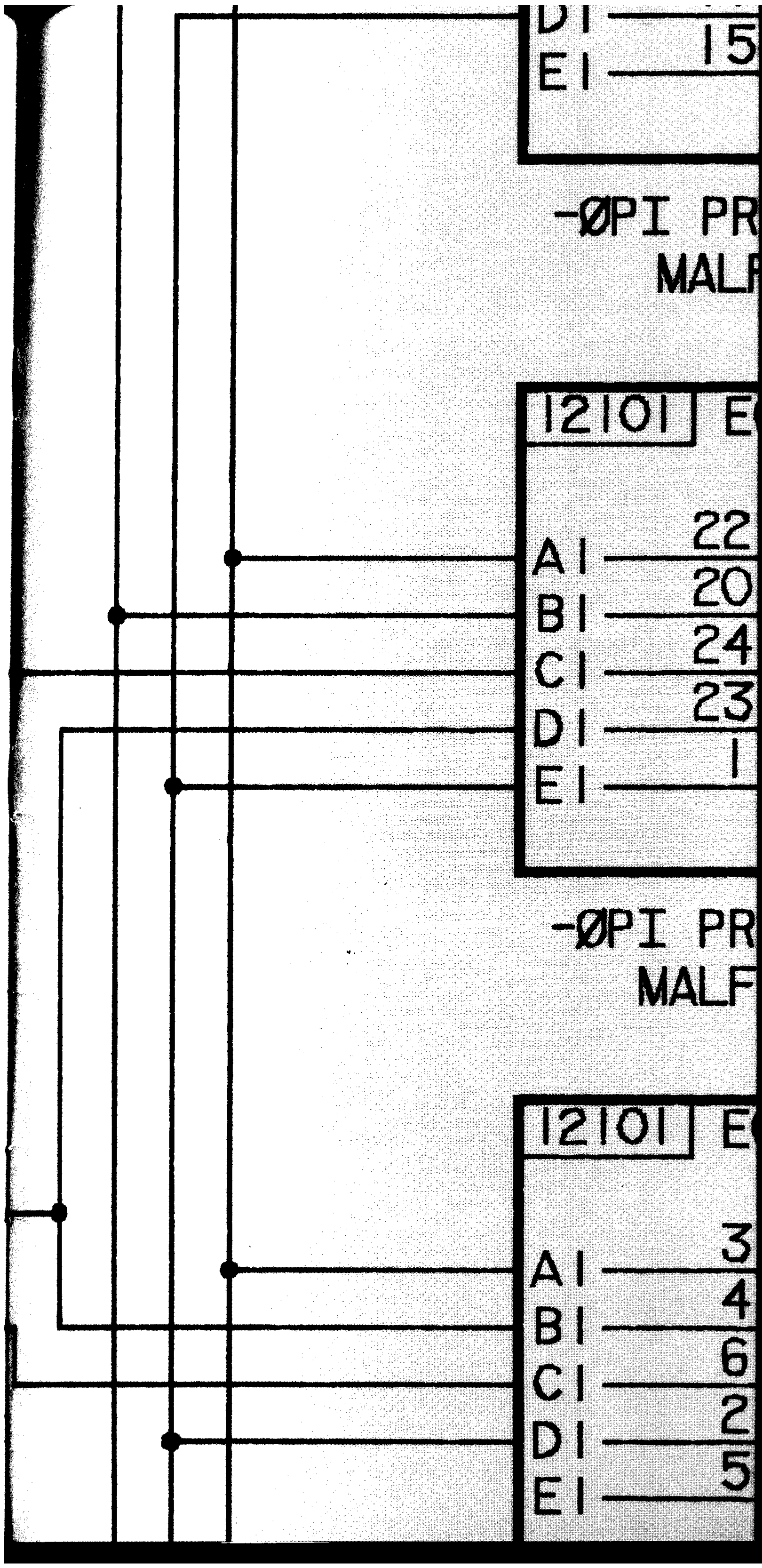
While the central processor is shared between 170 state and 180 state, other resources such as memory, peripheral processors, channels, and peripheral equipment are split between the states. Memory is statically partitioned with part assigned to NOS in 170 state and the rest assigned to NOS/VE in 180 state. The memory available to NOS can be further partitioned to include unified extended memory. (Figure 46.)

*For information on the instruction set, see Volume II of the hardware reference manual (referenced in the last chapter on Additional Sources of Information).

Figure 46



Users don't see the managing of both operating systems. They essentially log in to whichever system they want and then use that system as if it were the only one. Of course, users who are validated to do so can go back and forth between the two systems and transfer files.



Additional Sources of Information

Further information about the CYBER 180 series can be found in the hardware reference manuals. They can be ordered from:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, MN 55103

Volume I for each computer system contains the general system description and functional descriptions of each component of the system. This information varies from model to model. Volume II contains the central processor and peripheral processor instruction sets and programming information. With the exception of the vector instructions unique to the model 990, the information contained in this manual applies to every model. (The term virtual state used in the manual titles is synonymous with the term 180 state as it's been used in this overview.)

For 170 state information, the appropriate manuals are:

Manual Title	**Publication Number**
CYBER 170 Models 815 and 825 (CYBER 170 State) Hardware Reference Manual	60469350
CYBER 180 Models 810 and 830 (CYBER 170 State) Hardware Reference Manual	60469420
CYBER 170 Models 835, 845, and 855 CYBER 180 Models 835, 845, 855, and 990 (CYBER 170 State) Hardware Reference Manual	60469290

For 180 state information, the appropriate manuals are:

Manual Title	**Publication Number**
CYBER 170 Models 815 and 825 (Virtual State) Hardware Reference Manual Volume I	60469700
CYBER 180 Models 810 and 830 (Virtual State) Hardware Reference Manual Volume I	60469680
CYBER 170/180 Model 835 (Virtual State) Hardware Reference Manual Volume I	60469690
CYBER 170/180 Models 845 and 855 (Virtual State) Hardware Reference Manual Volume I	60461320
CYBER 180 Model 990 (Virtual State) Hardware Reference Manual Volume I	60462090
CYBER 170/180 Models 810, 815, 825, 830, 835, 845, 855, and 990 (Virtual State) Hardware Reference Manual Volume II	60458890

The CYBER 180 Computer Systems. For your information.

Today, Control Data is a $4.6 billion computer and financial services company with operations in 47 countries.

We attribute this success to the way we *apply* ourselves to solving problems for customers–to making certain that our systems and services meet your needs.

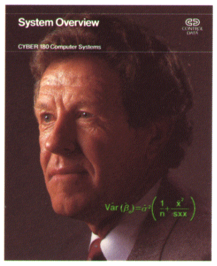
This dedication, along with our implementation of advanced technology, is the cornerstone of the CYBER 180 computer systems. They give you application and operating systems specifically designed to let you run what you need to run without outgrowing their capabilities. They're the systems for the 1990s–and beyond.

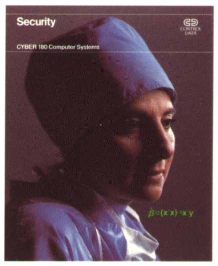
There are two ways to get additional information on the CYBER 180 computer systems, software, and peripherals.

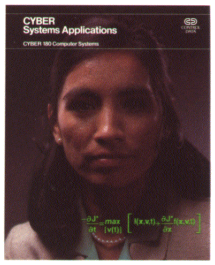
1. Contact your local Control Data representative.

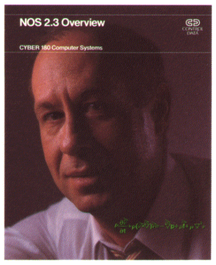
2. In the U.S.A., write Computer Systems, Control Data Corporation, P.O. Box 0, HQW09G, Minneapolis, Minnesota 55440.

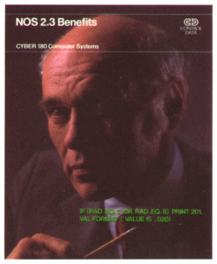
Outside the U.S.A., contact your local sales office or write Computer Systems, Control Data Corporation, International Operations, 7600 France Avenue South, ITL05F, Minneapolis, Minnesota 55435, U.S.A.

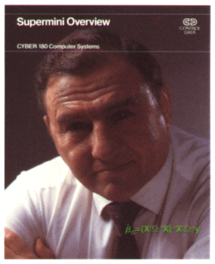


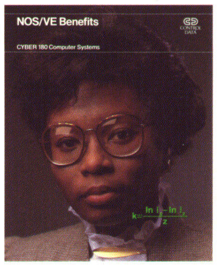


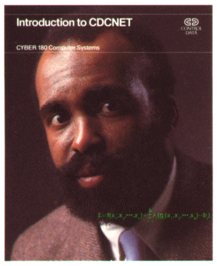


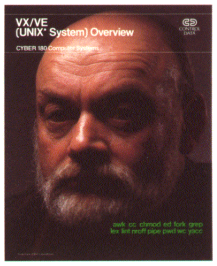


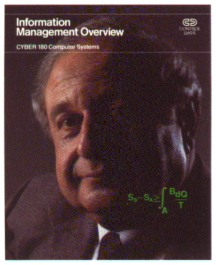












CONTROL DATA CORPORATION
POST OFFICE BOX 0
MINNEAPOLIS, MN 55440

SALES OFFICES AND SERVICE CENTERS
IN MAJOR CITIES
THROUGHOUT THE WORLD

CONTRO
DATA

204,137 LITHO IN U.S.A. 3/84

CD