GC28-1348-0
File No. S370-34

Systems

MVS/Extended Architecture Overview

First Edition (March, 1984)

This edition applies to:

MVS/System Product - JES2 Version 2 (5740-XC6)
MVS/System Product - JES3 Version 2 (5665-291)
MVS/XA Data Facility Product (DFP) (5665-284)

and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to makes these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program products may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department D58, Building 920-2, PO Box 390, Poughkeepsie, New York 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

The *MVS/Extended Architecture Overview* is an introduction to MVS/Extended Architecture (MVS/XA), the operating system that manages IBM System/370-XA computers. This book expects readers to have a general understanding of how computers work. That is, it assumes some background knowledge of the components of a computer system, the role of an operating system, and computer programming concepts.

Chapter 1 describes the environment in which MVS/XA runs and the attributes that allow MVS/XA to govern complex computer configurations. It presents MVS/XA interpretations of computer terms and concepts.

The remaining chapters give an overview of how MVS/XA works. Chapters 2 and 3 explain how MVS/XA manages and makes use of two key system resources: real storage and virtual storage. Subsequent chapters explain how MVS/XA accomplishes the primary operating system functions of:

- Chapter 4: Managing the hardware
- Chapter 5: Monitoring system resource use
- Chapter 6: Processing units of work
- Chapter 7: Reading and writing code and data
- Chapter 8: Identifying, tracking, and allocating resources to jobs
- Chapter 9: Preventing and tracing system errors
- Chapter 10: Recovering from system errors
- Chapter 11: Initializing the system

There are no prerequisite MVS/XA reading materials for using this book. Additional documentation of MVS/XA appears in the MVS/XA library. The *MVS/XA Library User's Guide*, GC28-1339, gives a complete list of related publications.

Contents

Figures

Chapter 1. Introduction to MVS/Extended Architecture

An operating system is a group of related programs that govern the computer system. The operating system controls the execution of programs and provides services they need to make use of the computer system hardware. MVS/XA is the operating system that takes advantage of the IBM System/370 extended architecture (System/370-XA).

A computer's architecture consists of the functions the computer system provides. It is distinct from the physical design, and, in fact, different machine designs may conform to the same computer architecture. In a sense, the architecture is the computer as seen by the user such as a system programmer. For example, part of the architecture is the set of machine instructions that the computer can recognize and execute.

System/370-XA, as its name implies, is an extension of the System/370 computer architecture. Similarly, MVS/XA is an extension of the MVS/370 operating system that supports the System/370 architecture. Thus, MVS/XA, despite the fact that it supports significant changes to System/370 architecture, includes much that is familiar to MVS/370 users.

The differences between MVS/370 and MVS/XA center on taking advantage of the continuing high performance enhancements to computer system hardware and improving the reliability, availability, and serviceability of the system. The two most significant changes are:

- 31-bit addressing

  MVS/370 provides a 24-bit addressing scheme. MVS/XA provides both a 31-bit and a 24-bit addressing scheme. This change extends the storage available to any one user from 16 million bytes (16 megabytes) to two billion bytes (two gigabytes).

- The channel subsystem

  The channel subsystem handles input and output (I/O) operations independently of the processors in the MVS/XA system. MVS/370 also allows overlap of I/O operations with instruction processing, but the MVS/XA channel subsystem increases the amount of overlap and allows all of the processors to access all of the I/O devices without the need for multiple tasks or for switches.

Neither of the changes, however, creates the need for users to change existing application programs. Application programs written for 24-bit addressing can run under MVS/XA, and there have been no changes to the way programs invoke I/O operations.

The MVS/XA Environment

To understand how and why MVS/XA functions as it does, it is important to understand the environment in which it functions. The special features that make MVS/XA unique reflect the features of the computer environments that MVS/XA manages.

By way of contrast, consider a simple, single user, computer system. Its operating system is a simple one that reads in one job, finds the data and devices it needs, lets the job run to completion, and then reads in another job.

The computer systems that MVS/XA manages are capable of **multiprogramming**, or executing many programs concurrently. By means of multiprogramming the system can, for example, run hundreds of jobs simultaneously for users who might be at distant geographical locations.

MVS/XA can also manage **multiprocessing**, which is the simultaneous operation of two or more processors that share the various system hardware devices. Figure 1-1 illustrates the IBM 3081 Processor Complex; two central processors share main storage and the channel subsystem. Chapter 4, "Multiprocessing" gives more detail on the MVS/XA multiprocessing environment.

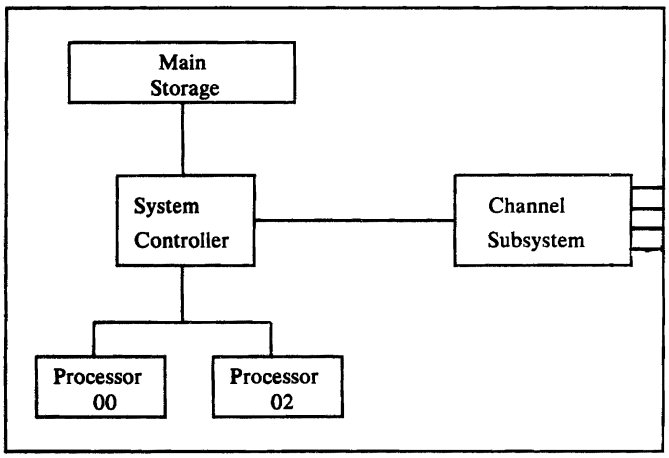


Figure 1-1. The 3081 Processor Complex

Many users running many separate programs means that, along with large amounts of complex hardware, MVS/XA users need large amounts of storage to ensure suitable system performance. They run sophisticated application programs that access large data bases and program modules. Such applications require the operating system to provide routines to protect privacy as well as routines for sharing the data bases and software services.

Thus, multiprogramming, multiprocessing, and the need for large amounts of storage mean that MVS/XA must provide function beyond simple job-to-job transition. The following introduction describes, in a general way, the attributes that enable MVS/XA to manage complex computer configurations. Subsequent chapters explain these features in more detail.

Virtual Storage

The MVS in MVS/XA stands for **multiple virtual storage** to indicate that **each user** has access to **virtual**, rather than only real (physical), main storage. Virtual storage means that each running program can assume it has access to all of the main storage that the addressing scheme allows. The only limit is the number of bits in a storage address. This ability to use a large number of storage locations is important because a program may be long and complex and, both the program's code and the data it requires must be in main storage in order for the processor to access them.

The 31-bit address supported by MVS/XA allows a program to address up to 2,147,483,648 (two gigabytes) storage locations. In contrast, the system has much less real storage. How much less depends upon the model of computer and the installation's configuration.

To allow each user to behave as though he had much more real storage than really exists in the computer system, MVS/XA keeps only the active portions of each program in real storage and the rest of its code and data in special data sets, usually on high-speed direct access storage devices (DASDs).

Virtual storage, then, is this combination of real and auxiliary storage. MVS/XA requires billions of bytes of auxiliary storage to make virtual storage possible. It uses a system of tables and bit settings to relate the DASD locations to real storage locations and keep track of the identity and authority of each program. Chapter 2, "Multiple Virtual Storage," explains how MVS/XA manages virtual and real storage.

Address Spaces

A complete two-gigabyte range of 31-bit virtual storage addresses is known as an **address space**. MVS/XA provides each user (batch job initiator, TSO user, or started task) with a unique address space and maintains the distinction between the code and data belonging to each address space. MVS/XA also includes **cross memory services**, that permit a single user to access other address spaces when necessary.

The ability of many users to share the same resources implies the need to protect users from one another and to protect the operating system itself. Along with such methods as "keys" for protecting real storage and code words for protecting data files and programs, separate address spaces ensure that users' programs and data do not overlap. Chapter 3, "MVS/XA Address Spaces," describes the virtual storage areas within each address space and which address spaces are created during system initialization.

Task Management

MVS/XA breaks each job into separate units of work known as tasks and attempts to process each one as efficiently as possible. The tasks for one job compete with one another, and with tasks related to other jobs, for use of system resources. Responsibility for controlling the progress of tasks through the system lies with the **supervisor**, a component of the operating system. The supervisor allocates resources (other than I/O devices) and maintains current information about each task so that processing can resume from the appropriate point in case of an interruption.

MVS/XA includes several mechanisms to enable the supervisor and other system components to maintain control. This section describes four control mechanisms: control blocks, the program status word, interruptions, and macro instructions. Chapter 6, "Supervising the Execution of Work," describes other key features of MVS/XA task management.

Control Blocks

MVS/XA modules normally store the information needed to control a particular unit of work or manage a resource in defined storage areas called control blocks. Generally speaking, there are three types of MVS/XA control blocks:

- System

  Each system-related control block represents one MVS/XA system. These contain system-wide information such as how many processors are functioning.

- Resource

  Each resource-related control block represents one resource such as a processor or auxiliary storage device.

- Task

  Each task-related control block represents one unit of work.

Control blocks work as vehicles for communication throughout MVS/XA. Such communication is possible because the structure of a control block is known to all of its users, and thus all can find needed information about the unit of work or resource. The MVS/XA system control blocks, for example, are all documented in the multi-volume *MVS/XA Debugging Handbook*.

Control blocks representing many units of the same type may be chained together on **queues**, with each control block pointing to the next one in the chain. A program can search the queue to find the data for a particular unit of work or resource, which might be:

- An address (of a control block or a required routine)
- Actual data, such as a value, a quantity, a parameter, or a name
- Status **flags** (usually single bits in a byte, where each bit has a specific meaning)

All fields in a control block are defined and identified in the documented structure of the specific control block.

Control blocks have many sizes and formats. Usually, a control block consists of a series of fullword fields, but some fields can be longer (such as the name of a data file) or shorter (such as a flag byte). Important points to remember about control blocks are that they are structured, documented, and usually chained together. Figure 1-2 illustrates a queue of task control blocks (TCBs).

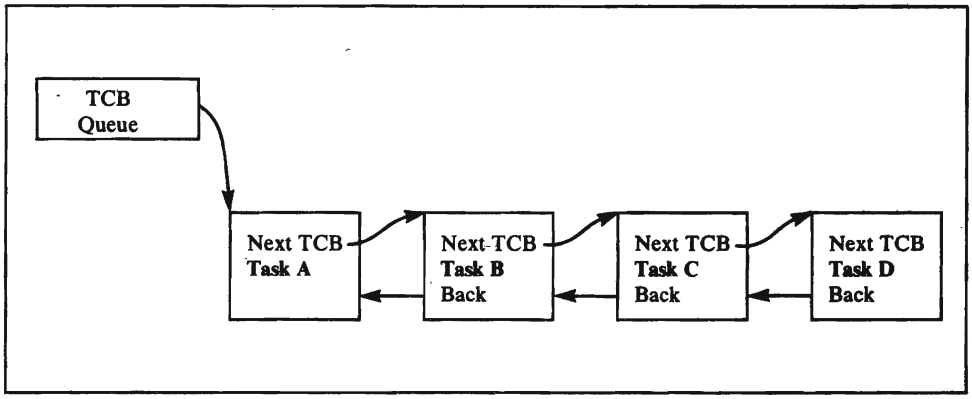


Figure 1-2. A Queue of Task Control Blocks

Program Status Word

The program status word (PSW) is a 64-bit data area in the processor that, along with control registers, timing registers, and the prefix registers, provides details crucial to both the hardware and the software. The current PSW includes the address of the next program instruction and control information about the program that is running, such as whether it is running in 24-bit or 31-bit addressing mode or whether or it is running in the **problem program** state or **supervisor** state.

Supervisor state programs are authorized to issue all instructions, including those that, for example, change the PSW. Problem programs may be IBM-distributed programs, such as language translators, or user-written application programs. They are not authorized to use all operating system instructions. Only when the problem state bit in the PSW is off can the program execute all instructions.

Each processor has only one current PSW. Thus, only one task can execute on a processor at any one time. Multiprogramming is possible, however, because an **interruption** causes the processor to save the contents of the current PSW and insert new PSW information in order to process the interruption. Figure 1-3 illustrates the MVS/XA PSW and some of its most important bits.

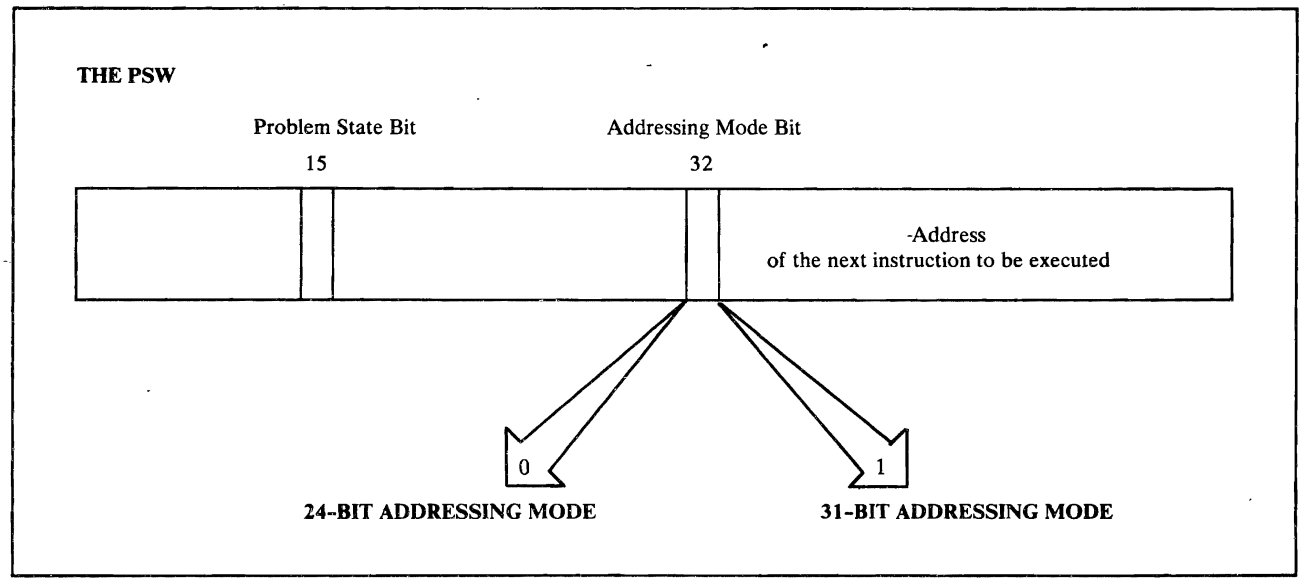


Figure 1-3. The MVS/XA PSW

Interruptions

An interruption is a request for attention from a processor. It indicates that an event, such as the completion of an I/O operation, expiration of a time interval, a program error, or a request for high-priority system services has taken place, and the system must reassess the mix of work to be done. When an interruption occurs, the processor temporarily ceases execution of the current task and begins executing an MVS/XA interruption handler.

First level interruption handlers (FLIHs) store the crucial information (such as the contents of the PSW) about the status of the interrupted task and give control to **second level interruption handlers** (SLIHs), which actually respond to the reason for the interruption.

After the interruption handlers complete their processing, a system component called the **dispatcher** might be called to select the highest priority ready unit of work (not necessarily the one that had been interrupted) and give it control until it completes or until another interruption occurs. An interruption thus allows the dispatcher to reassess the priorities of the tasks at hand.

Macro Instructions

Communication between MVS/XA programs occurs because system programmers follow established programming conventions and use common **macro** instructions. These instructions invoke segments of program code that map frequently used control blocks or perform frequently used system functions. MVS/XA macros, many of which are available to application programmers, exist for such functions as opening and closing data files, loading and deleting programs, and sending messages to the computer system operator.

Resource Management

Multiprogramming and multiprocessing create the need to measure the activity of the system and to adjust the workload to fit changing conditions. MVS/XA, for

example, monitors how much each active address space uses the processors, I/O devices, and real storage locations. The **system resources manager**, the MVS/XA component known as SRM, uses this information when it determines whether an address space should remain resident in real storage or whether a new address space should be created.

The system resources manager also takes into account the workload goals and priorities for users and equipment that the installation specifies in the installation performance specification (IPS). SRM, described in Chapter 5, "Managing System Resources," is the primary means by which the system and the installation manage the system resources.

MVS/XA provides other tools that allow the installation to control use of system resources. These include system parameters, exit routines, and the operator console.

System Parameters

System parameters are values specified by IBM or the installation and stored in the system data set named SYS1.PARMLIB. Each member of this data set contains parameters that the operator selects to control processing. For example, member IEASYS00 contains the default system parameters that tailor MVS/XA at start-up; the system uses these parameters and other values during the system initialization process.

Exit Routines

An exit is a defined point in system processing where a system program calls another program. The called program that IBM supplies performs standard default processing; it is designed to be replaced by a user version of the exit routine. The user exit routine performs user-defined functions appropriate to that particular point in MVS/XA processing.

Operator Console

The installation controls MVS/XA by entering **system commands** through one or more devices defined to MVS/XA as operator consoles. There is also a system console for use by IBM customer engineers in diagnosing and correcting hardware problems.

Through system commands issued at the operator console, the operator or system programmer can control MVS/XA or respond to a condition MVS/XA detects. System commands can:

- Change the status of hardware units, such as devices, between online (available) and offline (not available) to the system
- Monitor the status of various units of work in the system
- Change those system parameters that can be referenced after system initialization
- Start and stop system functions
- Set a trap for a recurring error condition

I/O and Data Management

Nearly all tasks involve some amount of data input or data output. The channel subsystem manages the use of I/O devices, such as disks, tapes, and printers, while MVS/XA, through software, associates the data for the task at hand with a device.

MVS/XA manages data by means of **data sets**. Data sets can hold information usually thought of as file data like the patients' records in a doctor's office. Or, data sets can hold information the computer needs, like parameters or programs.

The records in data sets may be organized in various ways, depending upon how the information will be accessed. Data sets can be organized for sequential access or direct access.

In a **sequential data set** organized for sequential access, records are data items that are stored consecutively. To retrieve the tenth item in the data set, for example, the system first passes by the preceding nine items. Data items that must all be used in sequence, like the alphabetical list of names in a seating chart, are best stored in sequential access data sets.

In a data set organized for direct access, also called random access, records are data items stored with control information so that the system can retrieve an item without searching all preceding items in the data set. Data items that are used frequently and in an unpredictable order are best stored in **direct access data sets**.

Partitioned data sets combine the features of sequential and direct access. The data set consists of a directory and **members**. The directory holds the address of each member and thus makes it possible to access each member directly. A member, however, consists of sequentially stored records.

Partitioned data sets are often called libraries. Programs are stored as members of partitioned data sets so that, even though they generally execute sequentially when running, the operating system can access them directly when selecting one for execution.

MVS/XA supports many different devices for data storage. Disks or tape are most frequently used for storing data sets on a long term basis. Disk drives are known as **direct access storage devices** (DASDs) because, even though some of the data sets on them might be stored sequentially, these devices can handle direct access. Tape drives are known as sequential access devices because data sets on tape must be accessed sequentially.

To enable the system to locate a specific data set quickly, MVS/XA includes a a data set known as the master catalog that permits access to any of the data sets in the computer system or to other catalogs of data sets. MVS/XA requires that the master catalog reside on a DASD that is always mounted on a drive that is online to the system. Certain other key data sets needed by the operating system reside on a particular DASD known as SYSRES, or the system residence volume, and must also always be on line. Chapter 7, "Satisfying I/O Requests," gives more details on how MVS/XA manages I/O operations and the transfer of data within the computer system.

Job Management

MVS/XA provides several ways to enter work into the computer system. With **batch processing**, a user enters a job through a local terminal or, by means of

remote job entry (RJE), through a remote terminal, or from tape, card reader, or disk, and the system processes the job at a later time. The operating system follows installation-defined guidelines as it chooses the time and resources for the job. With **interactive** job entry, such as the time sharing option (TSO), the system responds to terminal users while they are actually logged-on to the system. MVS/XA also permits the computer operator to enter a job by means of the START command; such jobs are called started tasks.

For MVS/XA, a job is more than the work to be done; it is the work to be done embedded in a stream of job control language (JCL) statements supplied by the user or the installation. JCL identifies such things as the system resources and data the job needs. The **job entry subsystem** (JES) processes the JCL, organizes the necessary programs, data, and resources, and presents MVS/XA with a job that is ready to be processed. Upon completion of the job, JES releases resources used for processing and schedules job output.

There are two IBM-supplied job entry subsystems: JES2 and JES3. Chapter 8, "Entering and Scheduling Work," describes how each one manages jobs.

Recovery Management

A data processing system must be available for use when it is needed. For a large system, this means that the system can function even if one component fails and can, possibly, diagnose the cause and correct or compensate for the failure. MVS/XA includes recovery mechanisms to prevent a user error from causing the failure of the computer system, to isolate and recover from operating system errors, and to protect the system from hardware errors. It also has programs that trace system activity and display the status and contents of various system resources. Chapter 9, "Monitoring System Activity," describes how MVS/XA monitors system activity; Chapter 10, "Recovering From Errors," describes the recovery mechanisms.

Summary

The operating system called MVS/XA is a combination of program and data modules. Large groups of modules that make a particular MVS/XA function possible are known as **system components**. Other groups of modules that provide added function that is dependent on MVS/XA are known as **subsystems**. MVS/XA includes a **subsystem interface** (SSI) for communication with IBM subsystems (such as the job entry subsystems) or user-supplied subsystems.

The motto, "divide and conquer", aptly describes how MVS/XA manages a computer configuration. MVS/XA gets work done by dividing it into pieces and giving portions of the job to components and subsystems that function interdependently. At any point in time, one component or another gets control of the processor, makes its contribution, and then passes control along to a user program or another component. There is no one entity that is MVS/XA. Rather, what exists is a collection of specialists acting according to accepted guidelines to get work done.

The remainder of this book describes important aspects of MVS/XA processing and gives an overview of what various components do and how they do it. It finishes with Chapter 11, "Initializing the System," which shows how components work together.

Chapter 2. Multiple Virtual Storage

The two gigabytes of storage in an MVS/XA address space are shared between user programs and MVS/XA system programs. System areas include the prefixed save area (PSA), which holds critical information unique to each processor in the system, the nucleus portion of the system control program that must always be in storage, and the commonly used system programs and subsystems. The map of an address space showing the addresses allocated for these areas is the same for all users. It is shown in Figure 2-1.

The organization of the MVS/XA virtual storage address space map arose from the need to maintain compatibility with programs written for MVS/370. Because of its 24-bit addressing scheme, MVS/370 provides address spaces with a maximum of 16 million bytes of virtual storage to be shared among user and system programs. Thus, maintaining compatibility means that MVS/XA must provide portions of each region at addresses below 16 megabytes as well as extended portions of these regions above the 16-megabyte line.

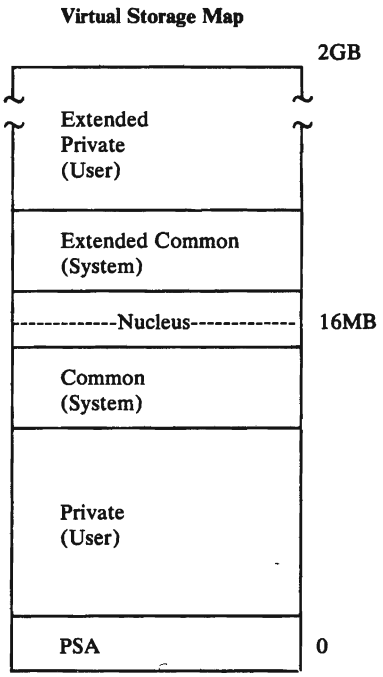


Figure 2-1. The MVS/XA Address Space

Addressing Mode and Residence Mode

To maintain compatibility with MVS/370, MVS/XA recognizes 24-bit addresses. Whether it interprets an address as 24 or 31 bit depends upon the setting of the addressing mode bit in the current PSW at the time an instruction executes. If this bit, bit 32, is set on, all addresses are interpreted as 31-bit addresses. Programs running in 31-bit mode can access locations zero to two gigabytes of virtual storage.

If the addressing mode bit is zero, the processor uses the 24 right-hand bits of an address. Programs running in 24-bit addressing mode can address the first 16 megabytes of virtual storage. MVS/XA allows programs to switch from one mode to another during execution in order to access data or call modules running in the

other mode. Thus, new programs can take advantage of 31-bit addressing and still be compatible with ones written for 24-bit addresses.

All MVS/XA program modules have an addressing mode (AMODE) attribute that indicates which addressing mode is to take affect when a module is given control. The AMODE attribute is assigned to an MVS/XA program module by the programmer, as input to the assembler or the linkage editor, or by default. The default is 24-bit addressing mode.

MVS/XA modules also have a residence mode (RMODE) attribute that indicates whether they must be loaded below the 16-megabyte address line or can be loaded anywhere in virtual storage. RMODE=24 modules require residency below 16-megabytes. RMODE=ANY allows the operating system to load a module anywhere in virtual storage.

A program that must be directly addressable by 24-bit callers must reside below the 16-megabyte line. A program that does not have 24-bit callers, or whose 24-bit callers call it indirectly, can reside anywhere. The RMODE attribute is assigned as input to the assembler or linkage editor, or established by default. RMODE=24 is the default residence mode.

Figure 2-2 shows the meaning of the AMODE and RMODE program attributes.

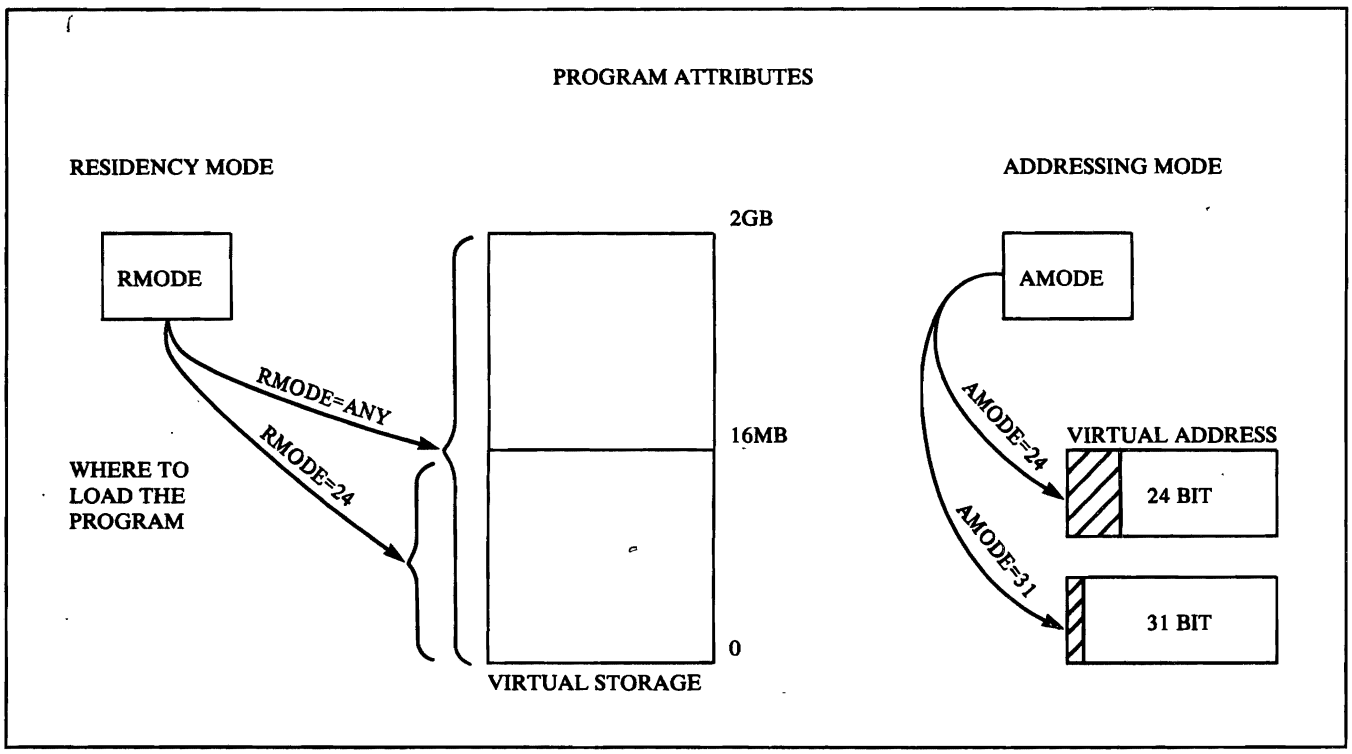


Figure 2-2. AMODE and RMODE Attributes

The AMODE and RMODE attributes can be assigned to modules in various combinations depending on the location of the code and data they use. Not all of the possible combinations make sense, however. The combination of AMODE=24 and RMODE=ANY, for example, is invalid because a program using 24-bit addresses cannot function in locations above the 16-megabyte line where more than 24 bits are needed to denote an address. The AMODE=ANY and RMODE=ANY combination can be specified, but the system translates it to

AMODE=31, RMODE=ANY at execution time. Figure 2-3 shows the possible combinations of program attributes and indicates which are ones are valid.

AMODE	RMODE	
	24	ANY
24	valid	invalid
31	valid	valid
ANY	valid	invalid

Figure 2-3. Valid AMODE and RMODE Combinations

Virtual and Real Storage

Virtual storage is the MVS/XA mechanism that makes it possible for a user to access the maximum amount of storage that can be addressed in 31 bits even though the system might have much less real storage. Virtual storage works because MVS/XA keeps active portions of each address space in real storage and inactive portions on high-speed DASD (auxiliary storage). It moves them back and forth as necessary to ensure that the program code and data for each user are in real storage when they are needed.

To enable the parts of a program in virtual storage to move between real storage and auxiliary storage, MVS/XA breaks real storage, virtual storage, and auxiliary storage into blocks:

- A block of real storage is a **frame**.
- A block of virtual storage is a **page**.
- A block of auxiliary storage is a **slot**.

A page, a frame, and a slot are all the same size: each is 4096 (4K) bytes. An active virtual storage page resides in a real storage frame; a virtual storage page that becomes inactive resides in an auxiliary storage slot.

Moving pages between real storage frames and auxiliary storage slots is called **paging**. Figure 2-4 shows how MVS/XA performs paging for a program that has been running in virtual storage. At point ❶, parts A, B, and C of a three-page program are in virtual storage. Page A is active and executing in a real storage frame, while pages B and C are inactive since they have been moved to auxiliary storage slots. At point ❷, page B is required; the system brings B in from auxiliary storage and puts it in an available real storage frame. At point ❸, page C is required; the system brings C in from auxiliary storage and puts it in an available real storage frame. If page A had not been used recently and the system needed its frame in real storage, page A would be moved to an auxiliary storage slot, as shown at point ❹.

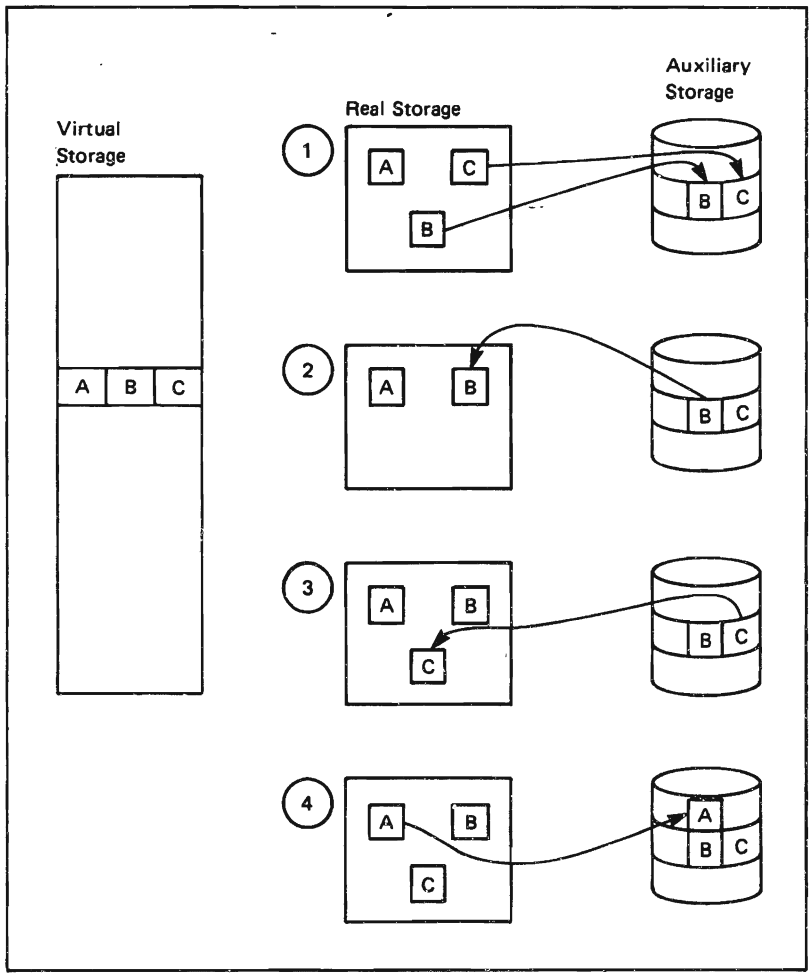


Figure 2-4. Virtual Storage Page Movement

Thus, the entire program resides in virtual storage; the system copies pages of the program between real storage frames and auxiliary storage slots to ensure that the pages that are currently active are in real storage as they are required. Note also that neither the frames nor the slots allocated to a program need to be contiguous; thus, a page could occupy several different frames and several different slots during the execution of a program. That is, if page A in the example became active again, MVS/XA would move it to any available frame.

Each address space competes with all other active address spaces for the use of real storage and other system resources, and the work being performed in each address space is paged between real and auxiliary storage. In order for this paging activity to take place quickly and efficiently, the system must be able to translate a virtual address (the address of a specific instruction or data item in virtual storage) into a real address (the address of the corresponding location in real storage). The solution is dynamic address translation.

Dynamic Address Translation (DAT)

Dynamic address translation (DAT) is a hardware feature that plays an important role in making virtual storage possible. The DAT hardware feature works with MVS/XA system software to translate a virtual address into a real address.

Virtual Address

In order to denote a location in virtual storage (create a virtual address), MVS/XA breaks the two gigabytes of virtual storage into 2048 segments, numbered 0 through 2047. Each segment consists of 1,048,576 bytes (one megabyte). The bytes in each segment are further broken down into 256 pages, numbered 0 through 255. Each page, as stated earlier, consists of 4K bytes. Within each page, a specific location is addressed by its byte displacement, that is, the number of bytes between the page origin and the specific location.

A virtual address, therefore, consists of the segment number, the page number within that segment, and the byte displacement within that page. Figure 2-5 shows how virtual storage is broken down to provide a 31-bit virtual address that consists of an eleven-bit segment number, an eight-bit page number and a twelve-bit byte displacement.

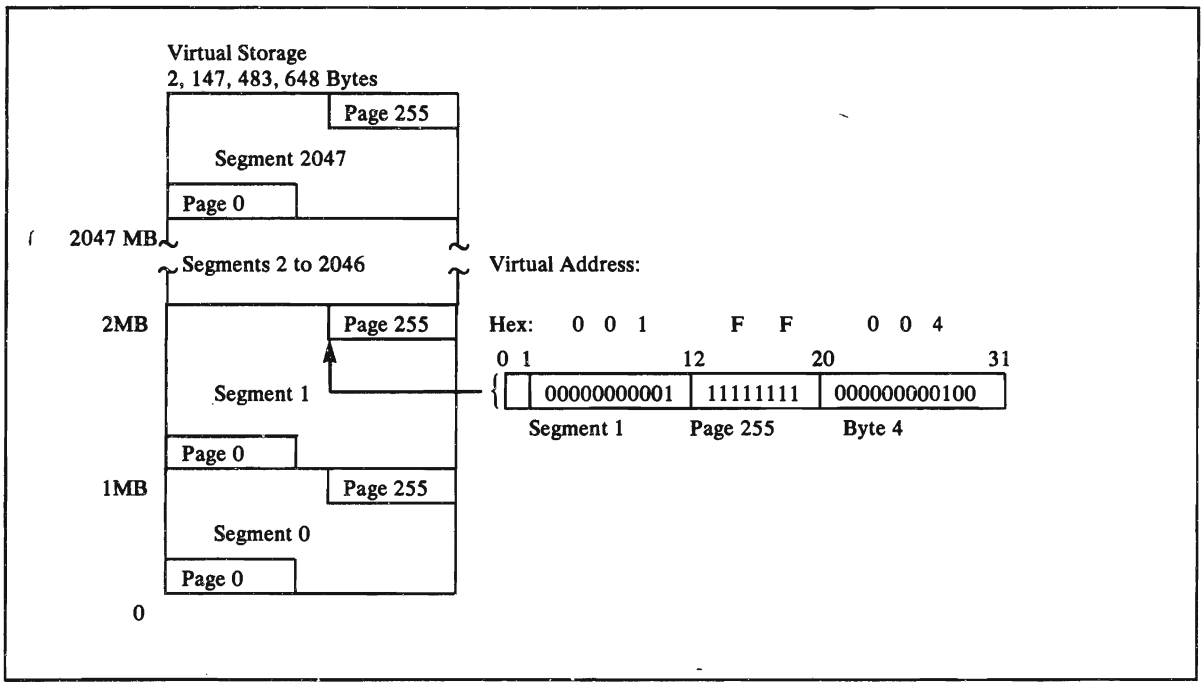


Figure 2-5. Virtual Storage Address

Segment and Page Tables

To translate a virtual address into a 31-bit real address, the DAT feature uses a control register, the **segment table origin register (STOR)** and one **segment table** and 2048 **page tables** for each address space. The segment table has one entry for each of the 2048 segments in the address space; each entry contains, among other things, a pointer to the page table for that particular segment. When address translation occurs, the STOR points to the segment table for the address space. This provides the distinction between a virtual address for one address space and the same virtual address for any others.

The page table for each segment has one entry for each of the 256 pages in the segment. If a page is currently in a real storage frame, the entry includes the page frame real address (PFRA) for the frame that corresponds to that page. If a copy of a page is not currently in a frame in real storage, the entry indicates this; the

invalid bit is set on, and the system must copy the page from auxiliary storage to real storage and update the page table before the virtual address can be successfully translated. The MVS/XA page table also contains a page protection bit that, when set, marks the corresponding frame as read-only. The system uses this bit to protect against unexpected modification of code and data. Figure 2-6 shows the relationship between the segment table, the page tables, and the pages in real storage.

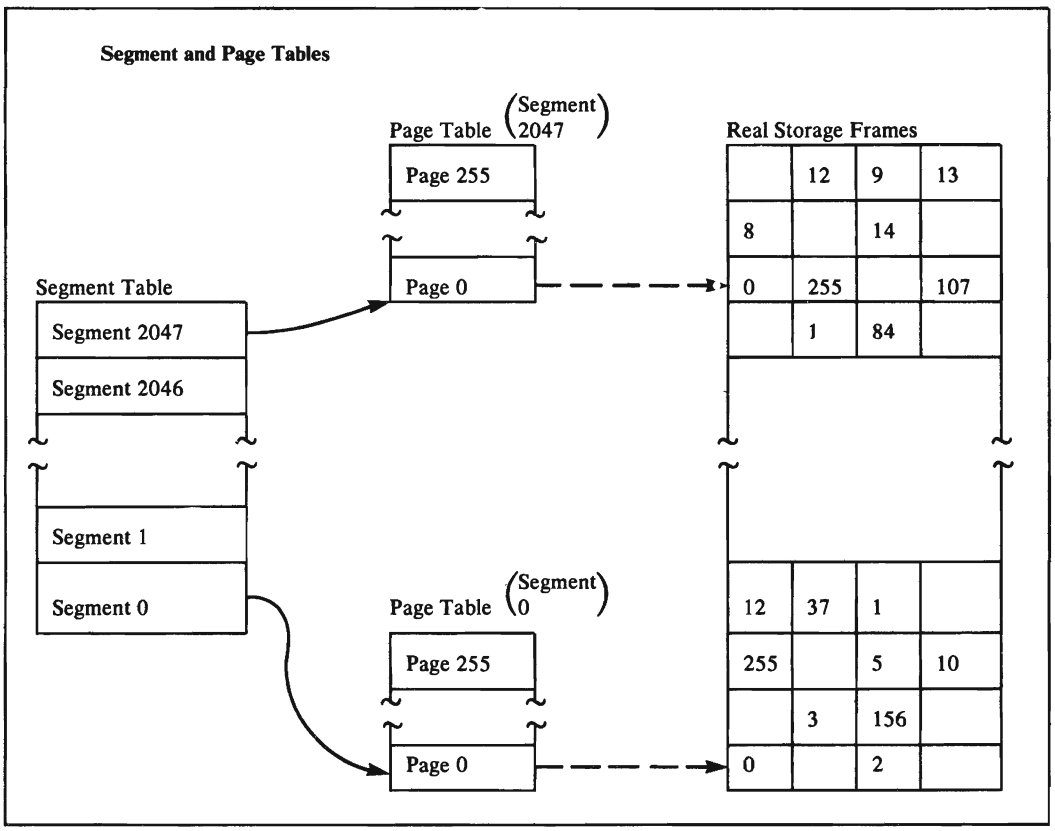


Figure 2-6. Segment Table and Page Tables

Two-Level Lookup

To translate a virtual address into a real address, DAT looks in two tables. Figure 2-7 illustrates this process. The first table lookup ❶ adds the address of the start of the segment table, in the segment table origin register (STOR), to the segment number from the virtual address (multiplied by 4 bytes, the length of each segment table entry) to locate the proper segment table entry. This entry contains the origin address of the page table for that segment. The second table lookup ❷ adds the page table origin to the page number in the virtual address (also multiplied by 4 bytes, the length of each page table entry) to locate the required entry in the page table. Unless the page is invalid, the page table entry contains the address of the real storage frame that holds the page specified in the virtual address. The final step ❸ in dynamic address translation adds the address of the real storage frame to the byte displacement within the frame. The byte displacement is the 12 rightmost bits of the virtual address. The result of this addition is the 31-bit real address.

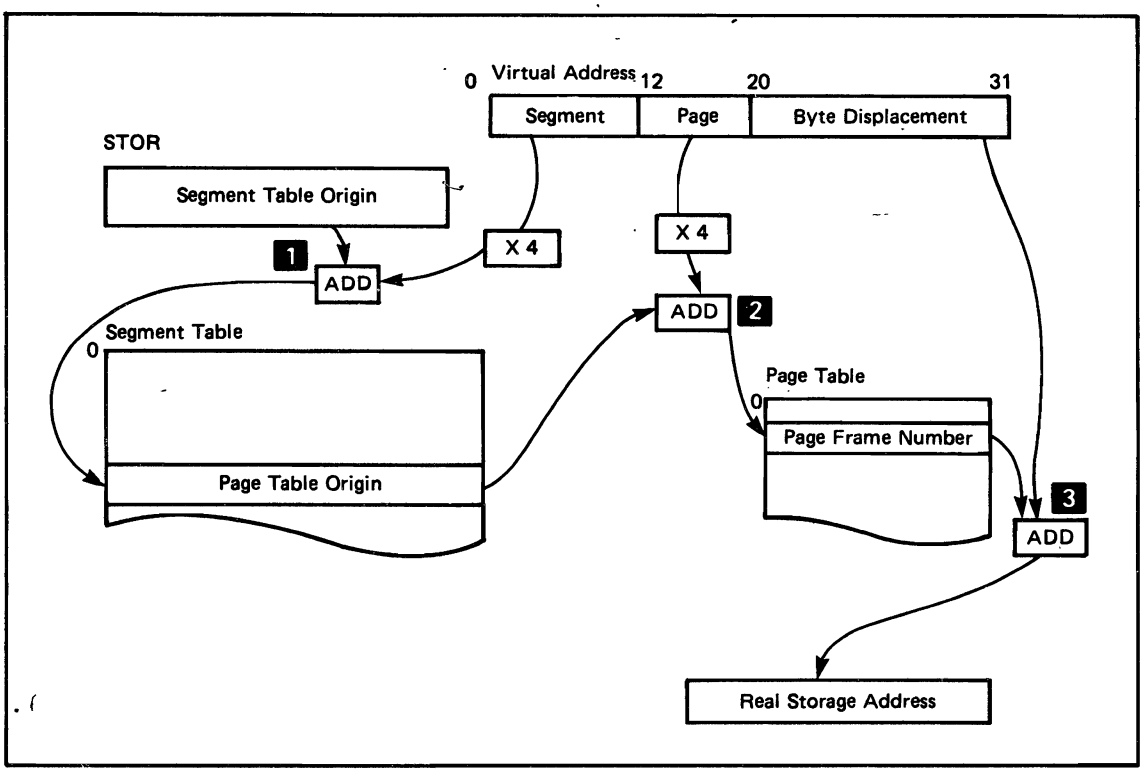


Figure 2-7. Dynamic Address Translation

Each time a virtual address is successfully translated into a real address, the system saves the address of the real storage frame in a special hardware buffer called the translation lookaside buffer (TLB). The TLB contains, an address space identifier, the segment number and page number from the virtual address, and the corresponding real storage address for the most active virtual pages. The DAT hardware checks the TLB before beginning the process of address translation, and, because a very high percentage of addresses can be found in the TLB, address translation time is significantly reduced by bypassing the two-level table lookup process most of the time.

When the first step of the table look up process encounters a segment table entry that has no corresponding page table in real storage, the DAT cannot translate the virtual address and a segment translation exception, or **segment fault** occurs. If the page table for the segment exists, paging is required to bring the page table into real storage. If the page table does not exist, one is built before paging occurs.

Similarly, when the second step of the table lookup process encounters an invalid page table entry, it means the required page is not in a real storage frame. The DAT hardware thus cannot translate the virtual address, and a page-translation exception, known as a **page fault**, occurs. If the page has been defined in the page table, **demand paging** - the transfer of a slot in auxiliary storage to a page in real storage on demand - is required to bring the page into real storage. If the page is not backed by a frame, a frame is assigned to the page and demand paging occurs.

The Paging Process

In addition to the DAT hardware and the segment and page tables required for address translation, paging activity involves a number of system components to

handle the movement of pages and several additional tables to keep track of the most current version of each page at any particular time.

To understand how paging works, assume that DAT encounters an invalid page table entry during address translation, indicating that a page is required that is not in a real storage frame. To resolve this page fault, the system must bring the page in from auxiliary storage. First, however, it must locate an available real storage frame. If there is no available frame, the request must be saved and an assigned frame must be freed. To free a frame, the system copies its contents to auxiliary storage and marks its corresponding page table entry as invalid. This operation is called a **page-out**. Actually, the system performs a page-out only when the contents of the frame have been changed since the page was copied into real storage. If the contents have not changed, the frame is freed by simply setting on the page table entry invalid bit.

After a frame is located for the required page, the contents of the page are copied from auxiliary storage to real storage and the page table invalid bit is set off. This operation is called a **page-in**. Actually, in order to avoid unnecessary I/O, the processor checks, before doing a page-in, to see if the frame that previously held the contents of the page has the same information and ownership as the slot on DASD indicating that the frame has not been changed. If so, the frame is **reclaimed** by setting the page table invalid bit off, and no actual data transfer occurs.

Figure 2-8 summarizes the paging process, showing how pages move between real and auxiliary storage in response to a page fault or to fill the need for an adequate supply of real storage frames.

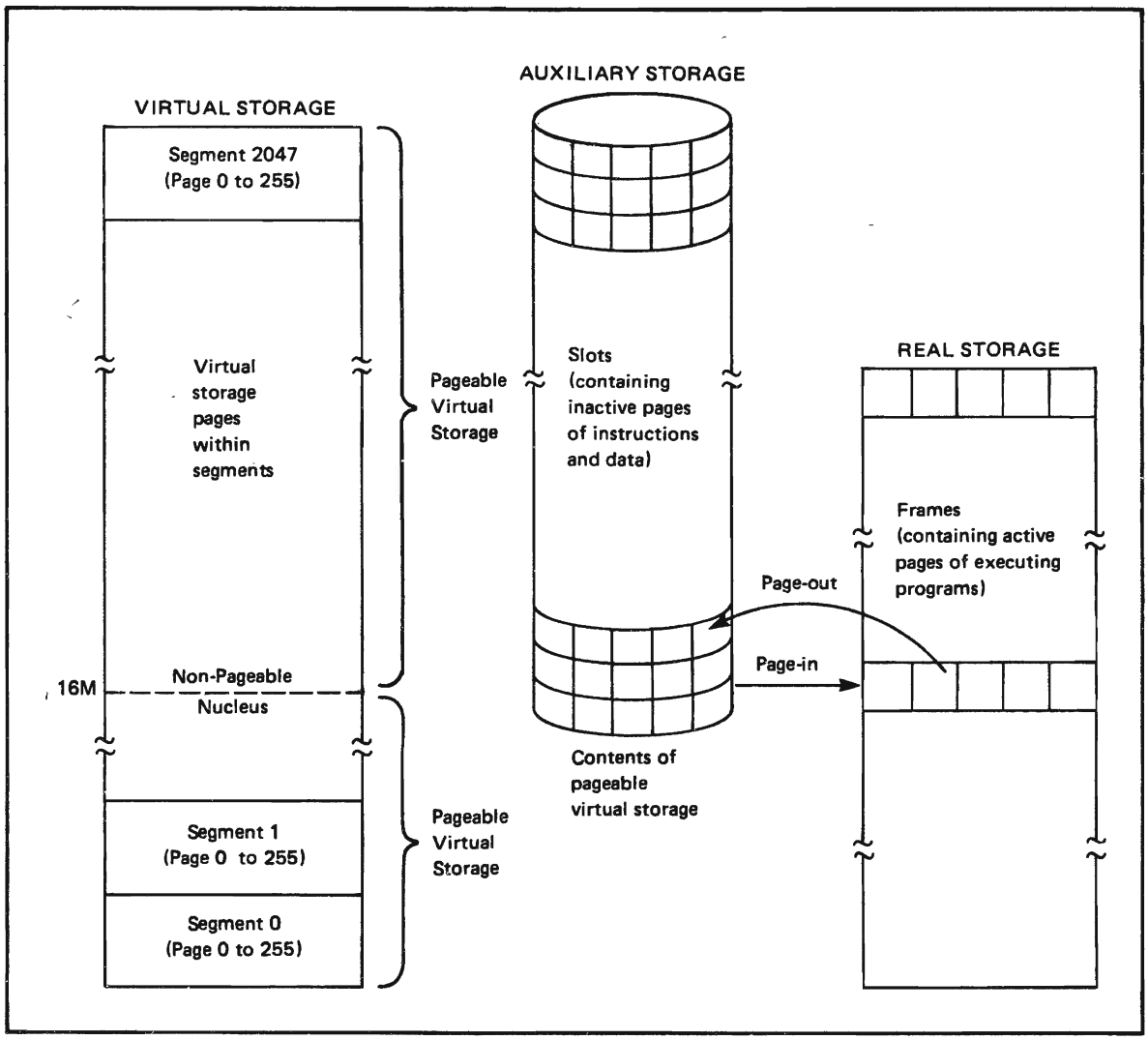


Figure 2-8. Page-Out and Page-In

Paging can also take place when the program loader loads a program into virtual storage. The program loader brings an entire program into virtual storage. MVS/XA obtains virtual storage for the user program, and allocates a real storage frame to each page. Each page is then active and subject to the normal paging activity; that is, the most active pages are retained in real storage while the pages not currently active might be paged out to auxiliary storage.

MVS/XA tries to keep an adequate supply of available real storage frames constantly on hand. When a program refers to a page that is not in real storage, the system uses a real storage page frame from a supply of available frames. When this supply becomes low, the system uses page stealing to replenish it.

Page Stealing

Page stealing occurs when the system takes a frame assigned to an active user and makes it available for other work. The decision to steal a particular page is based on the activity history of each page currently residing in a real storage frame. Pages that have not been accessed for a relatively long time are good candidates for page stealing.

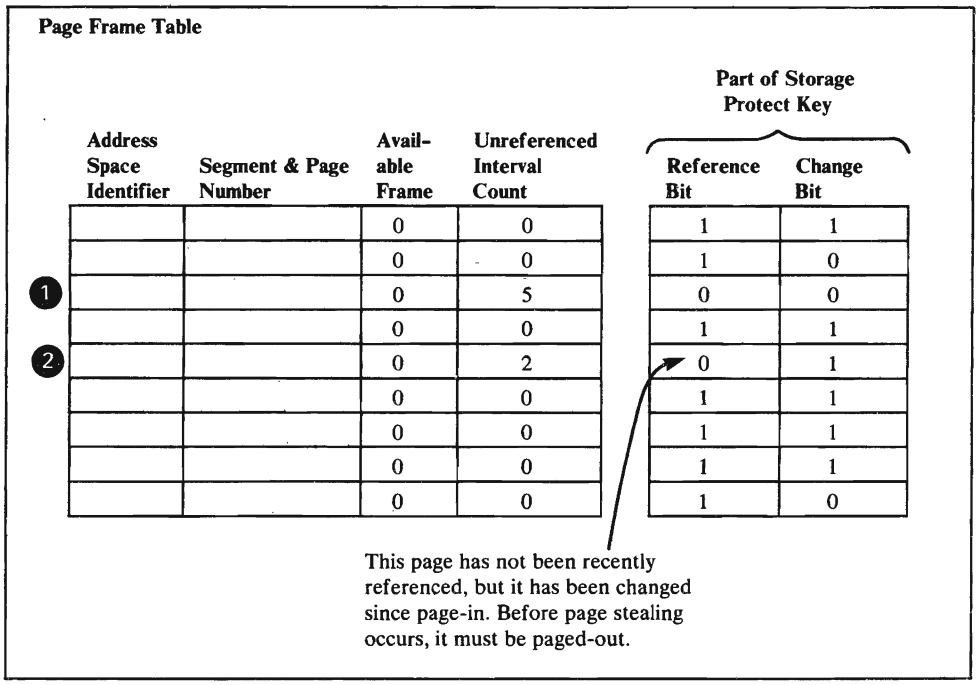
To determine which pages are to be stolen, MVS/XA examines the activity history of the pages that are currently in real storage. This information is held in the **page frame table**. There is one page frame table for the entire system, and it has an entry for each frame of real storage. Each entry identifies a page frame and includes the address space identifier and the virtual address within the address space for the page that is currently using the frame.

Other information in the entry describes the activity history of the page. The available frame field indicates whether the frame is currently in use or is available. Two additional bits associated with the frame, the reference bit and the change bit, are relevant when the frame is in use. (**Note:** These bits are actually part of a control field associated with each 4K block of storage. They are maintained by the hardware and used by the software to make paging decisions; they are therefore described here as if they were physically part of the page frame table.)

The **unreferenced interval count** indicates how long it has been since a program referenced the frame. The reference bit is set on by the hardware whenever a page frame is referenced. At regular intervals, the system checks the reference bit for each page frame. If the reference bit is not on -- that is, the frame has not been referenced -- the system adds to the page frame's unreferenced interval count. It adds the number of seconds since this address space last had the reference count checked. If the reference bit is on, the frame has been referenced and the system turns it off and sets the unreferenced interval count for the page frame to zero. Those page frames with the highest unreferenced interval counts are most likely to be stolen.

The change bit is set to zero when a page is initially brought into a real storage frame. When the contents of the page are changed the change bit is set on. Setting the change bit on tells the system that it must copy the contents of the frame to auxiliary storage before making the frame available for other work. Checking the change bit ensures that no changes made during program execution are lost during the paging process.

Figure 2-9 shows a portion of the page frame table and illustrates how the entries are set up and how the reference, change information, and unreferenced interval count are used to determine which pages will be stolen. All example frames are in use; the available frame bits are set to zero. The system checks the unreferenced interval count and finds two pages that have not been referenced recently. These two pages will be stolen. The first page ❶ has not been changed since it was brought in from auxiliary storage; therefore, no physical page-out is required to save its contents because the copy of the page in real storage is the same as the copy of the page in auxiliary storage. The second page ❷ has been changed; therefore the system performs a page-out before it steals the page, and the contents of the page are written to auxiliary storage. The system is thus able to steal two pages, only one of which requires a page-out. (The first page will be the first one selected for stealing because of its higher unreferenced interval count.)



	Address Space Identifier	Segment & Page Number	Avail-able Frame	Unreferenced Interval Count	Reference Bit	Change Bit
			0	0	1	1
			0	0	1	0
1			0	5	0	0
			0	0	1	1
2			0	2	0	1
			0	0	1	1
			0	0	1	1
			0	0	1	1
			0	0	1	0

Figure 2-9. Page Frame Table

Swapping

Swapping is the process of transferring all of the most recently valid private pages of an address space between virtual storage and auxiliary storage. This has the effect of moving an entire address space into or, out of, virtual storage. It is one of several methods MVS/XA employs to balance the system workload, as well as to ensure that an adequate supply of available real storage frames is maintained. Address spaces that are **swapped-in** are active, having pages in real storage frames and pages in auxiliary storage slots. Address spaces that are **swapped-out** are inactive; the address space resides on auxiliary storage and cannot execute until it is swapped in. Swapping is performed in response to recommendations from the system resources manager (SRM), described in Chapter 5, "Managing System Resources."

Storage Protection

Figure 2-4 showed how virtual storage works for one program; in reality, of course, many programs or users would be competing for use of the system. MVS/XA uses two techniques to preserve the integrity of each user's work: (1) a private address space for each user, as described in Chapter 3, "MVS/XA Address Spaces," and (2) multiple storage protect keys, as described in the following topic.

Storage Protect Keys

Under MVS/XA, the information in real storage is protected from unauthorized use by means of multiple storage protect keys. A control field in storage called a key is associated with each 4K frame of real storage. This field, or key, is not itself addressable except by special operating system instructions.

The key in storage contains the protect key that the user of the frame must have as well as a fetch protect bit. The **protect key** controls which, if any, users can modify the frame. (A bit in the page table, the **protection bit**, makes the frame read-only and thereby prevents modification by **any** user.) The **fetch-protect bit** also protects

a frame. When it is set, a program must have the same key as the frame or have key 0. Otherwise, it can neither modify the frame nor read, or fetch, its contents. Figure 2-10 shows the format of the key in storage.

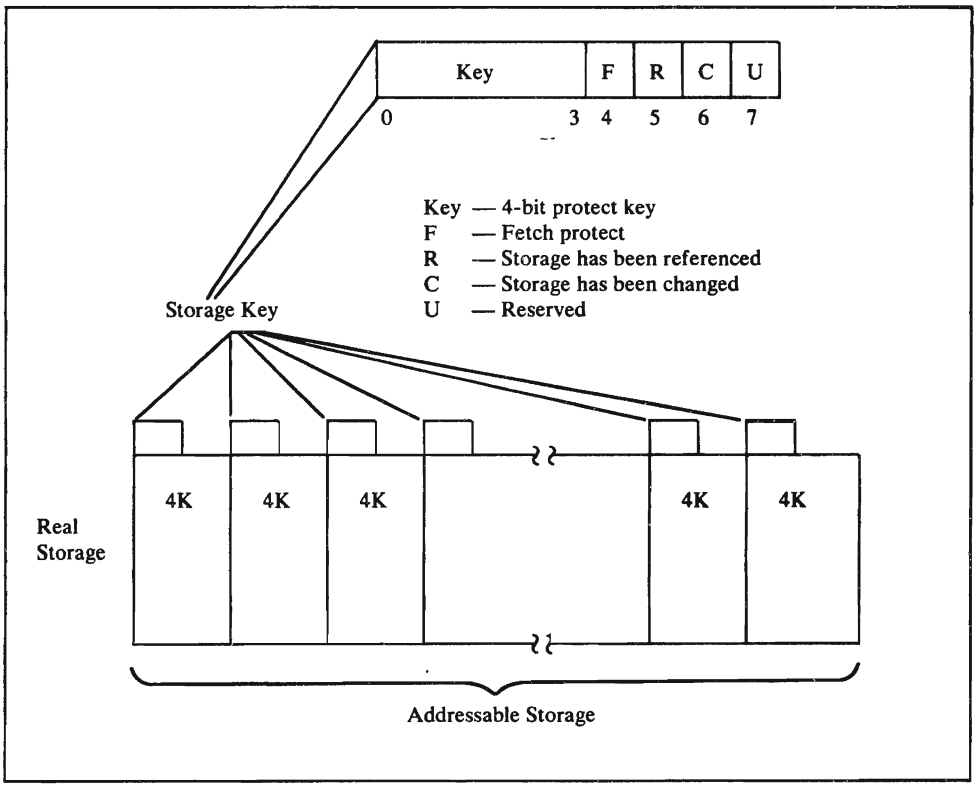


Figure 2-10. The Key in Storage

When a request is made to modify the contents of a real storage location, the key is compared to the storage protect key associated with the request, which appears in the current program status word (PSW). If the keys match or the program is executing in key 0, the request is satisfied. If the key associated with the request does not match the storage key, the system rejects the request and issues a program exception interruption.

When a request is made to read (or fetch) the contents of a real storage location, the request is automatically satisfied unless the fetch protect bit is on, indicating that the frame is fetch-protected. When a request is made to access the contents of a fetch-protected real storage location, the key in storage is compared to the key associated with the request. If the keys match, or the requestor is in key 0, the request is satisfied. If the keys do not match, and the requestor is not in key 0, the system rejects the request and issues a program exception interruption.

Key Assignments

There are sixteen possible storage protect keys available. A specific key is assigned according to the type of work being performed. Figure 2-11 summarizes the assignment of storage protect keys.

Storage protect keys 0 through 7 are reserved for the MVS/XA control program and various subsystems. Storage protect key 0 is the master key. Its use is restricted to those parts of the control program that require almost unlimited store and fetch capabilities. With two limitations, a storage protect key of 0 associated

with a request to access or modify the contents of a real storage location means that the request is satisfied. The limitations on the ability of key 0 to modify real storage are: first, no program can store into real storage locations 0 to 511; second, no program can store into real storage frames for which the page table protection bit is set on.

Storage protect keys 8 through 15 are assigned to users. Because all users are isolated in private address spaces, most users - those whose programs run in a virtual region - can use the same storage protect key. These users are called V=V (virtual=virtual) users and are assigned a key of 8. Some users, however, must run in a real storage region. These users are known as V=R (virtual=real) users and require individual storage protect keys because their addresses are not protected by the DAT process that keeps each address space distinct. Without separate keys, V=R users might reference each other's code and data. These keys are in the range of 9 through 15.

Key	Use
0	MVS/XA system control program
1	Job scheduler and job entry subsystems (JES2 or JES3)
2	Virtual Storage Personal Computing (VSPC)
3-4	Reserved
5	Data management
6	TCAM and VTAM
7	IMS
8	V=V users running in virtual storage
9-15	V=R users requiring real storage

Figure 2-11. Storage Protect Key Assignment

Key Switching

Frequently, a user program requests a service from a system (or subsystem) program; with the request the program passes the address of an area in storage to be modified by the system program. This area should belong to the user. However, if an error occurs and the area really belongs to the system instead of the user, the system could be destroyed. Thus, the system program does a key switch before performing the service for the user. A key switch means that the system program uses the storage protect key of the user program rather than its own storage protect key while performing the requested service.

MVS/XA Storage Managers

Real storage frames, auxiliary storage slots and the virtual storage pages that they support, are managed by separate components of MVS/XA. They are the real storage manager, the auxiliary storage manager, and the virtual storage manager.

Real Storage Manager (RSM)

The real storage manager (RSM) keeps track of the contents of real storage. It maintains the entries in the system's page frame table, and in each address space's page tables and associated **external page tables** that relate the virtual storage page to a page data set slot. It manages the paging activities described earlier such as page-in, page-out, and page stealing. RSM also assists with swapping an address space in or out, verifies the storage protect keys, and does **page fixing** (marking pages as unavailable for stealing).

Auxiliary Storage Manager (ASM)

The auxiliary storage manager (ASM) keeps track of the contents of the page data sets, the swap data sets and the VIO data sets (described in Chapter 7, "Satisfying I/O Requests"). Page data sets contain slots representing virtual storage pages that are not currently occupying a real storage frame. They also contain slots representing pages that do currently occupy a real frame but, because the frame's contents have not been changed, the slots are still valid.

Swap data sets contain the **working set** of an address space. Generally speaking, the working set is a subset of pages that were in real storage and associated with the address space when the swap out occurred. The working set includes the most recently referenced pages, pages fixed in real storage, and the segment and the page tables.

When a page-in or page-out is required, ASM works with RSM to locate the proper real storage frame and auxiliary storage slots. For a page-in, RSM reads the entries in the external page table to determine the slot location of a page, locates an available frame, and passes this information to ASM, which uses it to bring the slot into real storage. For a page-out, ASM locates an available slot on auxiliary storage, copies the page from real storage to auxiliary storage, and sends RSM the information needed to update the external page table.

Virtual Storage Manager (VSM)

The virtual storage manager (VSM) responds to requests to obtain and free virtual storage. It also manages storage allocation for any program that must run in real, rather than virtual storage. Storage is allocated to code and data when they are loaded in virtual storage. As they run, programs can request additional storage by means of the GETMAIN macro; they request the release of storage with the FREEMAIN macro instruction.

VSM keeps track of the map of virtual storage for each address space. In so doing, it sees an address space as a collection of 256 **subpools**. Subpools are logically related areas of virtual storage identified by numbers (0 to 255). Being logically related means the storage areas within a subpool share characteristics such as:

- Storage protect key
- Whether or not they are fetch protected
- Whether or not they are pageable
- Whether or not they are swappable

- Where they must reside in virtual storage (above or below the 16 megabyte line)
- Whether they can be shared by more than one task

Some subpools (with numbers 128 to 255) are predefined for use by system programs. Subpool 252, for example, is for authorized programs from authorized program libraries. Others (numbered 0 to 127) are defined by user programs.

Within an address space, VSM keeps track of:

- Unallocated areas:

  Virtual storage that is not allocated to a subpool

- Allocated areas:

  Virtual storage that is allocated to a subpool

- Free areas:

  Virtual storage within a subpool that is not being used

Chapter 3. MVS/XA Address Spaces

Conceptually, an MVS/XA address space consists of the two gigabytes of virtual storage available to each user. Figure 3-1 shows an address space as the rectangular description of virtual storage.

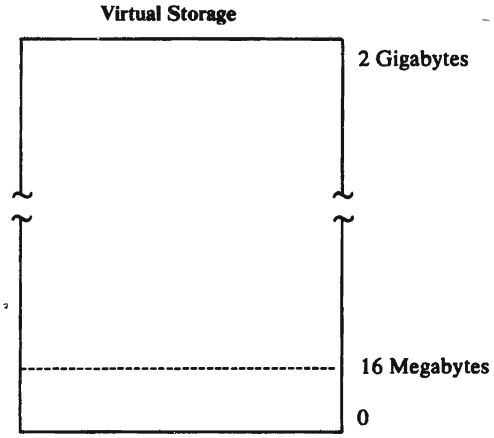


Figure 3-1. A Logical Representation of Virtual Storage

An MVS/XA address space contains the system prefix save area, private areas, and common areas. Each user has an entire address space and thus has access to all three kinds of areas. MVS/XA effectively isolates one address space from another by means of segment and page tables. Through the common areas of the address space, users can share programs and data areas. Thus, MVS/XA balances both the need to share resources and the need to maintain users' privacy.

Virtual Storage Areas

Program modules and data are located within an address space according to characteristics such as whether:

- They can be shared among all address spaces
- They can be paged or must always be backed by real storage (fixed)
- They must reside below the 16-megabyte line

The mapping of an MVS/XA address space in Figure 3-2 shows the various areas of an address space. It appears as it does because of the need to maintain compatibility with MVS/370. Almost every area exists below the 16-megabyte line and has an extended area above the line. As much as possible, MVS/XA treats each area of virtual storage below the line and its extended portion above 16 megabytes as one logical area. For example, if you request a report (dump) of the contents of the common service area (CSA), the system dumps both the CSA below 16 megabytes and the extended CSA. The sections that follow describe the areas of the virtual storage map.

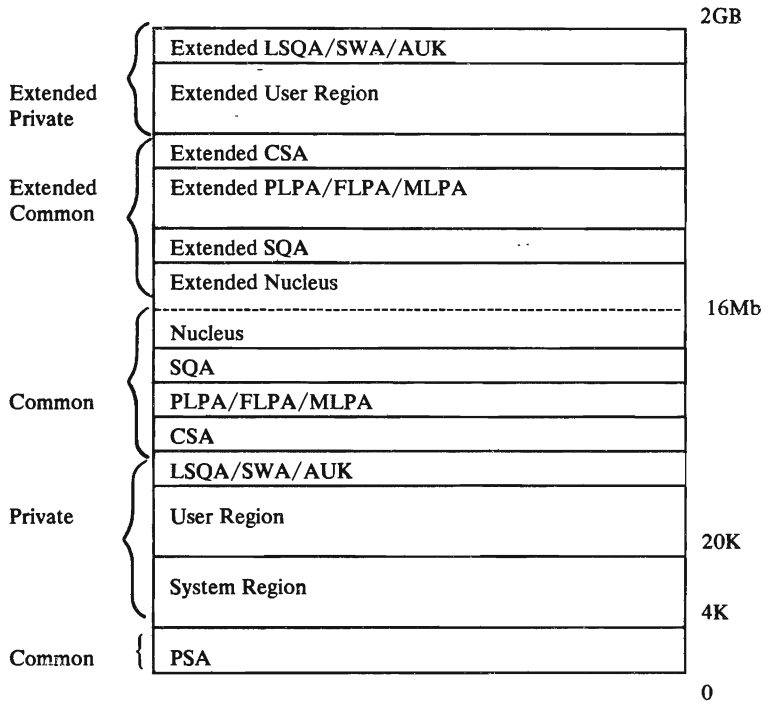


Figure 3-2. Virtual Storage Layout

Prefixed Save Area

The PSA contains critical information about both the MVS/XA operating system and the processor. It includes fixed storage locations for such things as the data items that become the contents of the current PSW when an interruption occurs, register save areas for system routines, and pointers to important control blocks. It is always fixed in real storage and never paged out.

For a uniprocessor, the PSA occupies the first 4K, the first page, of virtual and real storage. Each processor in a multiprocessing system running MVS/XA also addresses its own PSA as though the PSA were fixed in the first 4K of storage. MVS/XA uses the **prefix register** and a technique called **prefixing** to distinguish the PSA of one processor from the PSA of another.

With **prefixing**, the processors do not use absolute locations 0-4095. Rather, each processor has its own separate PSA and its own prefix register. When a processor is brought on line, the real starting address of its PSA is stored in its prefix register. Whenever the processor uses an address between 0 and 4095, the hardware adds the the contents of the prefix register to the address and uses the result. With prefixing, the address that normally would be the absolute address of the information in the first page of storage becomes an offset from the start of the real PSA. Because each processor's prefix register contains a different address, each processor can address locations 0 to 4095 and reference its own data.

The Private Area and Extended Private Area

The private area contains modules and data not shared by other address spaces. It consists of five sections:

1. System region
2. User region/extended user region

3. Authorized user key (AUK)/extended AUK
4. Scheduler work area (SWA)/extended SWA
5. Local system queue area (LSQA)/extended LSQA

The last three areas (AUK, SWA, and LSQA) are intermixed in the private area virtual addresses and are separate from the system region and the user/extended user regions.

System Region

The system region is the only section of the private area that does not have a counterpart above the 16M line. It is used by system functions performing work for an address space. These system functions run under the **region control task (RCT)**. The region control task is the highest level task in each address space; it plays a key role when an address space must be swapped in or out. The system region consists of four virtual pages (locations 4K to 20K) allocated from the bottom of the private area.

User Region/Extended User Region

The user region is the section of the private area in which user programs run. MVS/XA programmers try to use the extended user region as much as possible because it is vastly larger than the user region below the 16-megabyte line.

There are two types of user regions: virtual (V=V) and real (V=R). The two types are mutually exclusive; that is, a user region can be V=V or V=R, but it cannot be both.

A virtual (V=V) user region can be any size up to the size of the private area minus the size of LSQA, SWA, AUK, and the system region. Its size can be limited by the REGION parameter on the user's JOB or EXEC statement or by installation-written program exits.

V=V user regions are pageable and swappable. Only enough real storage frames are allocated at any particular time to hold the recently accessed parts of the user program.

Real (V=R) regions occur only below the 16 megabyte line. Each virtual address in the region always corresponds to the same real address. Figure 3-3 illustrates V=R storage mapping. Real storage for the entire region is allocated and fixed when the real region is created. Thus, a V=R job is non-pageable and non-swappable.

The installation must reserve sufficient real storage for all V=R regions that might exist at any one time. During system generation, the REAL= parameter of the CTRLPROG macro reserves real storage; during system initialization, the REAL= system parameter establishes real region storage limits. The system uses storage in the V=R area for normal paging activity if the V=R storage is not being currently used for V=R jobs. Particularly when system activity is high, a V=R job might not be started immediately; it must wait until the system can free the storage the V=R job requires.

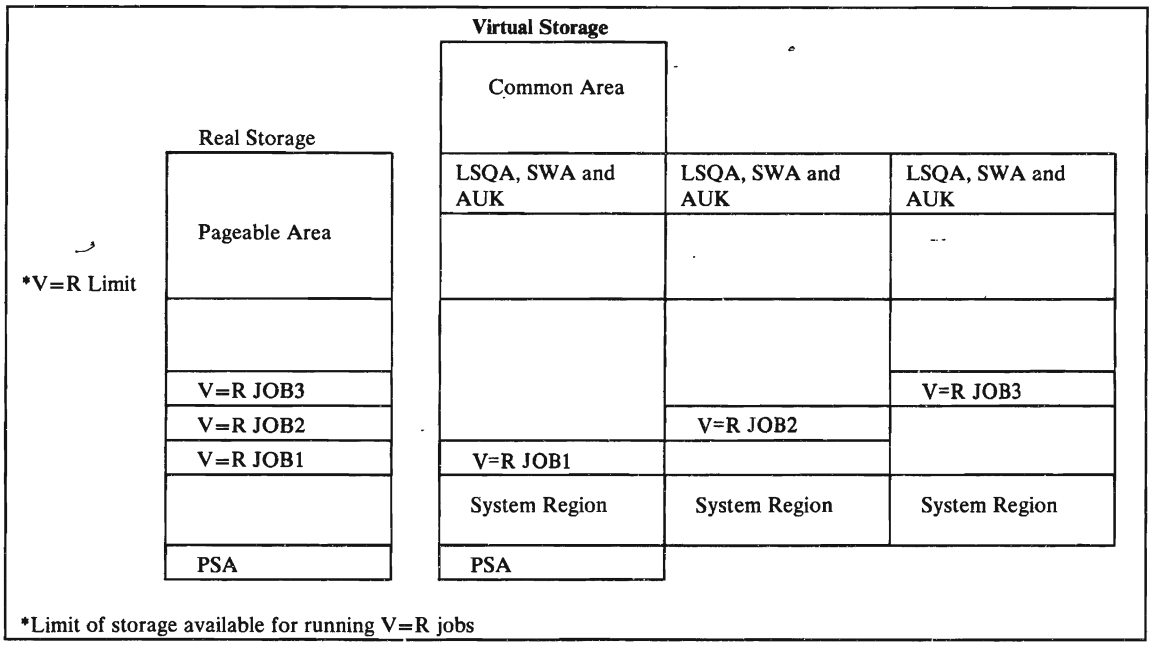


Figure 3-3. V=R Storage Mapping

Real regions should be used only for jobs with time-dependent functions (that is, jobs that cannot wait for paging activity to take place) or for jobs that cannot run in the virtual environment, such as jobs with channel programs that use the program control interruption (PCI) to modify themselves dynamically. See, "Satisfying I/O Requests," later in this book for more information about channel programs.

The default V=R region size is controlled by the VRREGN parameter in the IEASYSxx member of SYS1.PARMLIB. It can be overridden by the REGION parameter in a user JOB or EXEC statement.

Authorized User Key (AUK)/Extended AUK

The authorized user key (AUK) area of the private region contains system data relating to a specific user. Protected user resources, such as the data extent block (DEB) that describes a user data set, reside in this area.

This area is also identified as subpools 229 and 230. Subpools 229 and 230 are both protected by the user's storage key, that is, by the key in the PSW that is associated with the program using the storage. In addition, subpool 229 is fetch-protected, which means that its contents cannot even be read unless the key in storage matches the key in the PSW.

The AUK also contains data for the LNKLST lookaside (LLA) directory of modules in the system's LNKLST libraries. Because this directory is always in storage, it provides fast access to system modules and reduces I/O operations that consume time and channel paths.

Scheduler Work Area (SWA)/Extended SWA

The scheduler work area (SWA) contains the control blocks that exist from job step initiation to job step termination. These contain the internal form of the job control language (JCL) statements that accompany a job. The information in SWA is created when a job is interpreted and used during job initiation and execution.

(Chapter 8, "Entering and Scheduling Work," describes how MVS/XA processes a job.) The SWA is pageable and swappable.

Local System Queue Area (LSQA)/Extended LSQA

The local system queue area (LSQA) contains tables and queues that are unique to a particular address space. For example, LSQA includes the user's segment table and private area page tables. LSQA also contains all the control blocks that the region control task (RCT) requires. LSQA is swappable but not pageable. That is, the LSQA for each address space that is swapped-in is fixed in real storage frames.

The Common Area and Extended Common Area

The common area holds system information, such as program code, control blocks, tables, and data areas. It is common to all address spaces in the sense that any type of data or code in this area has the same virtual addresses in all address spaces. The common area includes:

1. Common service area (CSA) and extended CSA
2. Pageable link pack area (PLPA) and extended PLPA
3. Fixed link pack area (FLPA) and extended FLPA
4. Modified link pack area (MLPA) and extended MLPA
5. System queue area (SQA) and extended SQA
6. Nucleus and extended nucleus

Common Service Area (CSA)/Extended CSA

The common service area is addressable by all active programs and is shared by all swapped-in users. Data associated with an individual address space can be isolated by a storage protect key, but the primary advantage of CSA is to enable inter-address space communication.

CSA contains some fixed and some pageable system and user data areas; pageable areas are paged in and out of real storage as required. The total amount of storage for CSA is specified during system initialization and is allocated in 4K pages.

Pageable Link Pack Area (PLPA)/Extended PLPA

The pageable link pack area contains MVS/XA control program functions (SVC routines), access methods, other read-only system programs, and selected user programs. Because these modules are heavily used, and loading the PLPA is a lengthy process, MVS/XA normally saves its contents from one start-up to another.

As its name implies, PLPA is pageable; however, no physical page-outs are performed. PLPA pages that have not recently been used, however, might be stolen.

PLPA space is allocated in 4K pages. The size of PLPA is determined by the number of modules included, and, once the size is set, PLPA does not expand dynamically.

Fixed Link Pack Area (FLPA)/Extended FLPA

FLPA pages are fixed in real storage. They contain modules that could be in PLPA but require the extremely fast response that comes from having fixed pages.

Because FLPA is fixed, it reduces the amount of real storage available for other uses, such as running installation programs. Thus, the modules selected for FLPA are chosen with care. The MVS/XA paging routines normally keep a heavily-used PLPA module in real storage. Therefore, the most likely candidates for FLPA are modules that are infrequently used (those whose pages would be stolen) but require rapid response when they are used. An installation determines the size and contents of the fixed link pack area each time the MVS/XA system is started.

Modified Link Pack Area (MLPA)/Extended MLPA

The modified link pack area can be used for reentrant modules from selected system or user libraries; it acts as an extension to PLPA, but exists only for the duration of the current MVS/XA session. The system does not save the contents of the MLPA from one MVS/XA start-up to another as it does for the PLPA.

MLPA modules are normally read-only. Because MVS/XA searches the MLPA before it searches the PLPA, installations often use the MLPA to test modules before adding them to the PLPA.

System Queue Area (SQA)/Extended SQA

The system queue area (SQA) contains tables and queues that relate to the entire system. For example, the page tables that define the system area and the common area reside in SQA. The contents of SQA depend on an installation's configuration and job requirements.

The installation specifies the amount of storage for SQA when the system is initialized. If MVS/XA needs more storage for SQA, it uses CSA storage. If the system then runs out of CSA, it stops creating address spaces. The SQA is always fixed in real storage.

Nucleus/Extended Nucleus

The nucleus and the extended nucleus hold the resident part of the MVS/XA control program. Aside from the control program load module, the nucleus and extended nucleus contain the page frame table entries (PFTEs), data extent blocks (DEBs) for the system libraries, recovery management support routines, and unit control blocks (UCBs) for the I/O devices. The nucleus and extended nucleus surround the 16-megabyte line in virtual storage. They actually comprise what is known as the **DAT-on nucleus**. The hardware DAT feature translates their addresses to real addresses.

MVS/XA, however, also includes a **DAT-off nucleus** that consists of modules that must operate with the DAT feature off. It includes such routines as the recovery processing that occurs when a hardware problem makes the DAT feature inoperable. The DAT-off nucleus resides in real storage. It uses the highest real addresses available at the time it is loaded. The DAT-off nucleus is not part of virtual storage.

The DAT-on nucleus is fixed in real storage and is divided into four sections as shown in Figure 3-4. These are the read-only and read-write sections for both the nucleus and extended nucleus. While the size of the DAT-on nucleus varies depending on the system configuration and the extensions and options an installation chooses, the size of the nucleus does not change as additional jobs are swapped in and out.

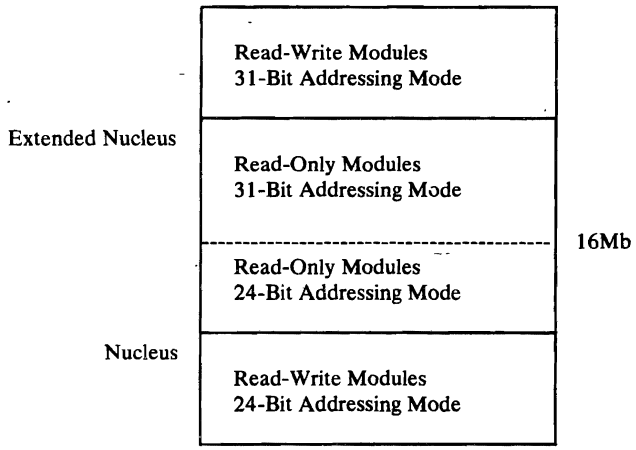


Figure 3-4. The DAT-on Nucleus

MVS/XA System Component Address Spaces

The master scheduler, a component of MVS/XA, interacts with operator commands and system parameters to initiate required functions. For example, the master scheduler controls the creation of address spaces.

When an MVS/XA system is initialized, the master scheduler address space is the first one created. Then, key system component address spaces are created. Because the master scheduler address space is the first address space, initializing its common areas also initializes the common areas for all address spaces. Thus all system components with their own address spaces have access to the following areas:

1. The private area below the 16-megabyte line
2. The common area, which surrounds the 16-megabyte line and includes the nucleus.
3. The extended private area above the 16-megabyte line
4. The prefixed save area (PSA), which resides at location 0

A system component can execute in the address space of a function that requests its services, or, if it has one, it can execute in its own address space. By creating its own address space to hold some or all of its data and executable code, a system component can reduce the amount of storage it requires in the common areas of virtual storage. If a system component has its own address space, that address space must be created and initialized to be capable of handling the requests of other address spaces.

The following system components have address spaces created during system initialization. They are listed in the order of their creation.

1. Program call/authorization (PC/AUTH address space) PC/AUTH is the first system component address space initialized. The PC/AUTH initialization routines initialize all the cross-memory tables needed to establish communication with other address spaces. As other system component address spaces are established, their associated initialization routines use PC/AUTH services to create and initialize their own cross-memory tables.

2. System trace (TRACE address space).

3. Global resource serialization.

4. Dumping services (DUMPSRV address space).

5. Communications task (CONSOLE address space).

6. Allocation (ALLOCAS address space).

7. System management facilities (SMF address space).

8. Primary job entry subsystem (JES2 or JES3 address space).

9. LNKLST lookaside (LLA address space).

Figure 3-5 shows the layout of the storage areas for the system address spaces that are created during system initialization.

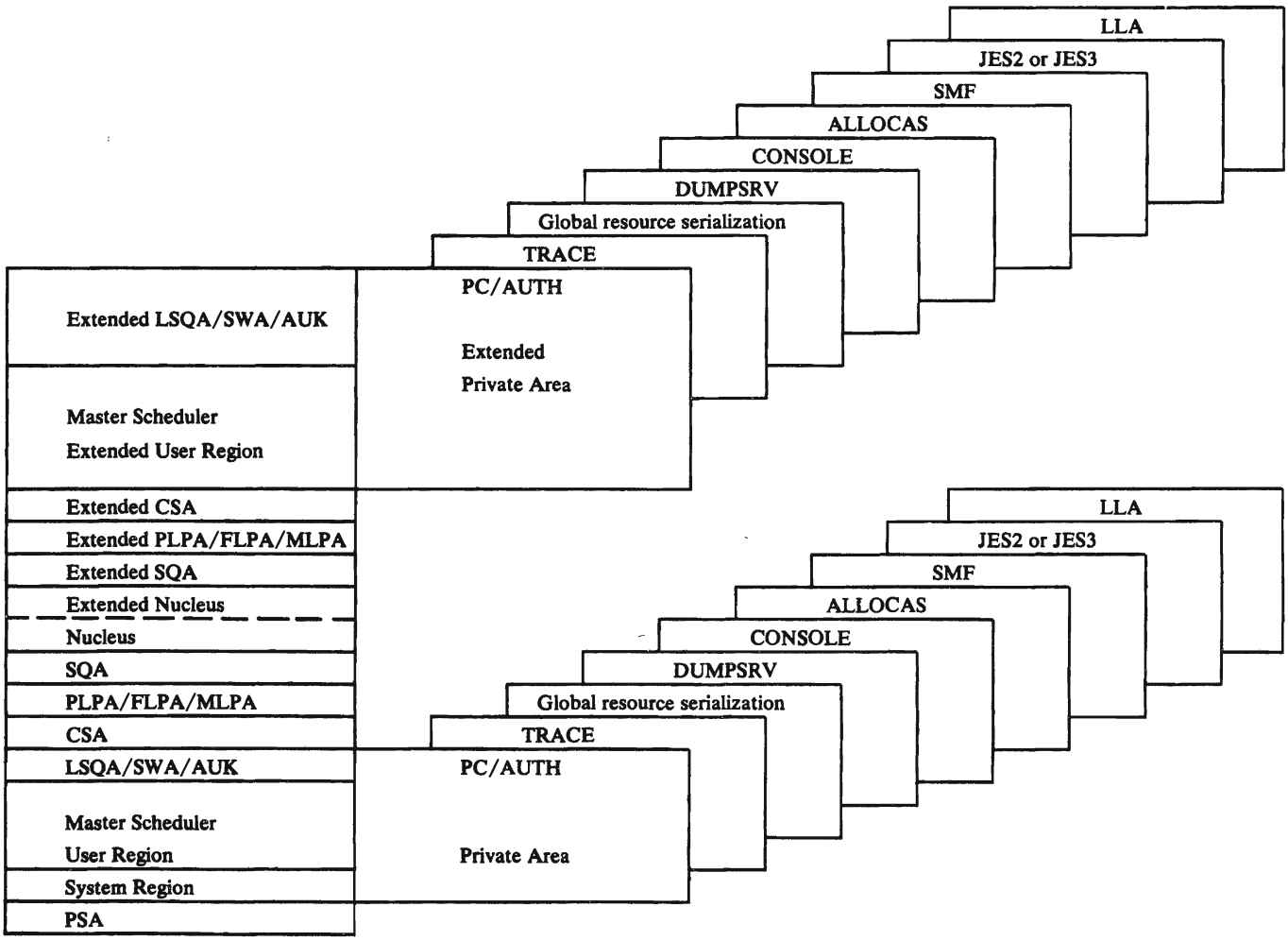


Figure 3-5. Virtual Storage Layout For Key MVS/XA Components.

Inter-Address Space Communication

There are two ways to communicate between address spaces: asynchronously and synchronously. Asynchronous inter-address space is controlled by control blocks known as service request blocks (SRBs) and is explained in Chapter 6, "Supervising the Execution of Work." The synchronous form of inter-address space communication is known as **cross memory**.

Cross Memory

Cross memory allows programs to pass control to programs in other address spaces and to move data from one address space to another. Because a program using cross-memory capabilities can directly access programs and data in the private area of another address space, cross memory can reduce the amount of common area needed in the virtual address spaces in the system. By using macro instructions to define a program as having cross-memory capability it is possible to control access to the shared data.

Chapter 4. Multiprocessing

Multiprocessing (MP) provides the solution to the need for increased computer power and increased computer system availability. Uniprocessing (UP) is the traditional starting point for a computer installation, but, as applications grow and online users proliferate, a single processor often becomes inadequate.

A uniprocessor is a single-processor system that contains its own main storage, is controlled by a single operating system, and has no direct communication with other processors. If it needs repair or maintenance, it must be removed from service.

In contrast, a multiprocessing system has at least two processors that share at least some of the system's resources. These processors can interchange tasks and subtasks to maintain a steady flow of work. One processor might initiate an I/O operation and another might handle the interruption that occurs when it has completed. If one processor fails, another is usually available to pick up its work and carry on.

Types of Multiprocessing

There are two types of multiprocessing:

- Loosely-coupled multiprocessing, where processors operate under separate operating systems yet share access to data such as a common workload queue. The processors are connected by shared DASD or by channel-to-channel (CTC) adapters and by shared DASD.

- Tightly-coupled multiprocessing, where at least two processors operate under the control of a single operating system. Some tightly-coupled multiprocessing systems consist of processor configurations that can be divided in half (partitioned) to form two independent configurations. Other tightly-coupled systems, known as a **dyadic** multiprocessor systems, cannot be partitioned.

Loosely-Coupled Multiprocessing

Loosely-coupled multiprocessing affords an easy growth path. The installation can connect many combinations of UP or MP systems into a single configuration with the following traits:

- The processors share a common workload queue.
- Each processor has its own operating system
- Jobs can, if necessary, be routed to a particular processor

Figure 4-1 illustrates a loosely-coupled system.

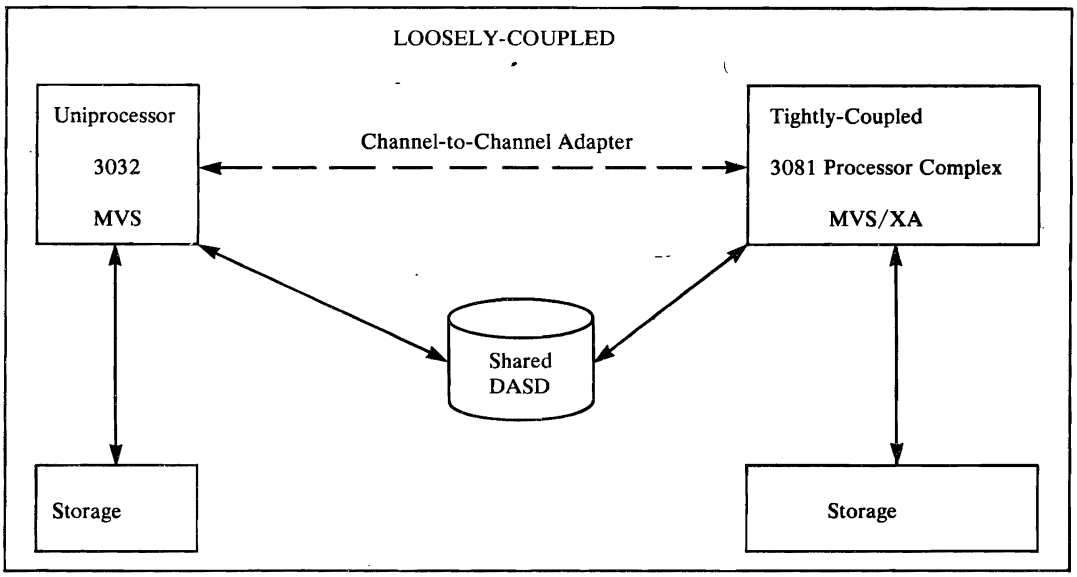


Figure 4-1. Loosely-Coupled Processing

Tightly-Coupled Multiprocessing

In a tightly-coupled multiprocessing system, the two or more processors share main storage, communicate directly with each other, and operate under the control of a single operating system. The MP system presents a single system image to the operator. The operator needs to communicate with and control only one operating system. Even though there are two or more processors available for work, the operator has one operational interface to the entire system, one job scheduling interface, and one point of control for all the resources available. The operator also can dynamically change the hardware configuration to meet various needs and control the operation of the processors and yet keep their individual control and status information separate.

Configuring a Tightly-Coupled Processor

A tightly-coupled MP configuration consists of many hardware components. **Reconfiguration** is the process of adding or removing some of these components from the configuration. The reconfigurable components in a system are:

1. Processors
2. Channel paths
3. Storage
4. I/O devices

Reconfiguration is usually initiated for one of three reasons:

1. A component, such as a segment of storage or a channel path, has malfunctioned and is interfering with the operation of the system. Depending on the circumstances, either the operator or the system can initiate the reconfiguring of the failing component offline so the system can continue processing without it.

2. One or more components are scheduled for maintenance

3. In larger systems, a change in workload necessitates the reconfiguring of a single system into two separate systems. In this case, the operator would

configure the appropriate processors, channel paths, storage, and devices offline from the running system and configure them into the second system. Figure 4-2 shows the 3084 Processor Complex, which usually runs with four processors. The illustration shows sides A and B, which are each composed of two processors. The 3084 Processor Complex can be reconfigured to become two independent MP systems with two processors each.

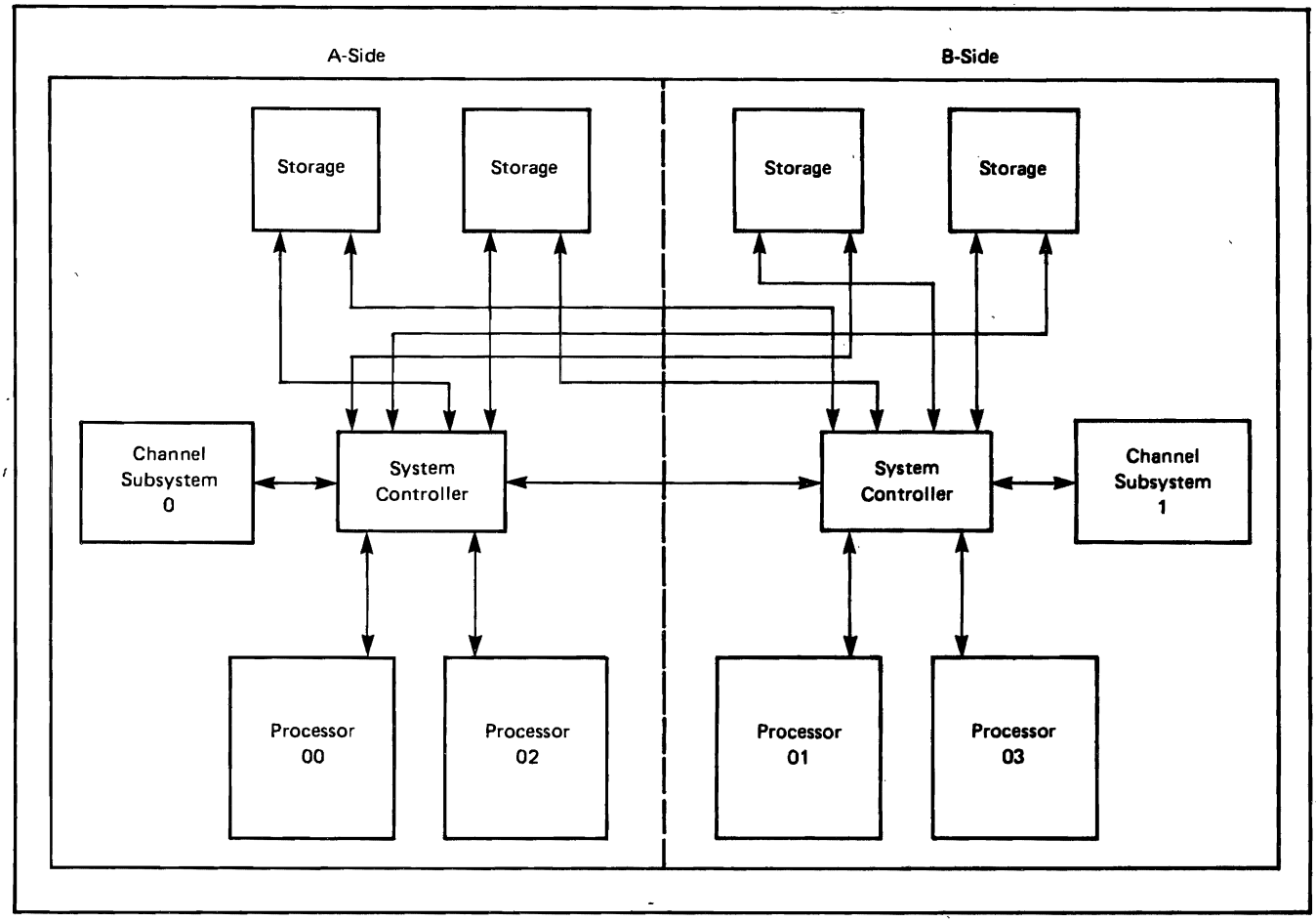


Figure 4-2. Tightly-Coupled Multiprocessing

The operator initiates reconfiguration with the MVS/XA CONFIG and VARY commands, specifying which elements to configure and whether they are to be made online or offline to the system. Reconfiguration processing has two stages:

1. **Logical reconfiguration**, which makes the component online or offline to MVS/XA. This process involves marking entries in MVS/XA system resource tables.

2. **Physical reconfiguration**, which makes the component online or offline to the hardware. This process often involves the setting of hardware switches that control whether access to the component is physically possible.

Both logical and physical reconfiguration are performed or initiated by the MVS/XA reconfiguration command processor. The CONFIG command also allows the operator to display which hardware components are presently online to

the system and which items are available to be configured online or offline. The DISPLAY M command provides the status of the hardware components.

Dyadic Tightly-Coupled Multiprocessing

A dyadic processor consists of two processors sharing storage and the channel subsystem. The processors are coordinated by a system controller that monitors communication and controls data flow between the two processors. Although there are two processors, the dyadic processor cannot be reconfigured into two uniprocessors. Figure 4-3 illustrates a 3081 processor complex, which is a dyadic processor.

If one processor fails in a dyadic multiprocessing system, which cannot be reconfigured into separate systems, the work of the failed processor is switched to the operative processor. The operator can remove the failing processor from the configuration and continue processing (with some performance degradation) on the remaining processor. However, repair of the failing processor must wait until the entire processor complex can be shut down.

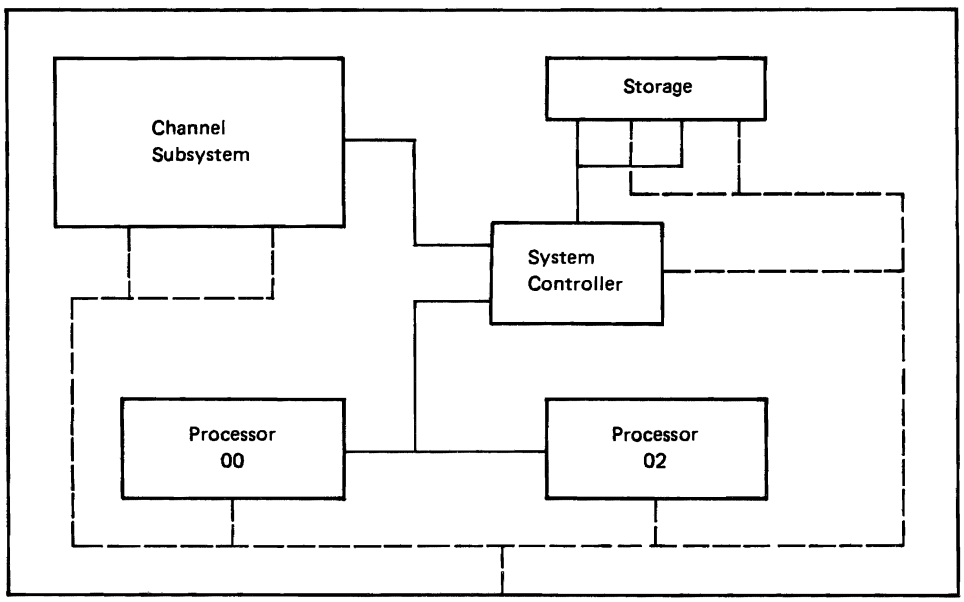


Figure 4-3. Dyadic Processor Complex

Control of Processing in a Tightly-Coupled MP System

Although tightly-coupled MPs share all real storage and run under the control of a single MVS/XA operating system, each processor must have a unique physical address for identification purposes. Likewise, each processor must have its own status and control information.

As explained in Chapter 3, "MVS/XA Address Spaces," the hardware and MVS/XA software maintain status and control information in specifically-assigned real storage locations called the prefixed save area (PSA). Each processor views the PSA as a 4096-byte block of fixed storage in the low-address range (storage locations 0-4095) of real storage. However, because multiprocessors can execute more than one job simultaneously, (one in each processor) each processor has its own PSA and uses a prefixing technique, also described in Chapter 3, to access its own PSA.

Communication Among Processors in an MP System

To control the system resources, the processors in an MP system must communicate with each other. Communication between the processors is called interprocessor communication (IPC). The MVS/XA software and system hardware both provide support for IPC.

MVS/XA-Initiated Communication

MVS/XA establishes interprocessor communication for several purposes:

- To perform system initialization
- To dispatch work
- To stop or restart a processor during reconfiguration
- To attempt alternate CPU recovery (ACR)

To accomplish this communication, MVS/XA uses the **signal processor** (SIGP) instruction. A SIGP instruction signals a processor and transmits a request to perform a function. The addressed processor decodes the request, performs the requested function (if possible), and transmits a response to the calling processor. The response contains a condition code and status information.

Some of the conditions that cause SIGP requests are:

- Initialization

  During the initialization of a tightly-coupled MP system, MVS/XA can determine whether other processors are online by issuing a SIGP sense instruction to each of the other processors. Each processor responds with an indication of its status. If the response indicates the processor is online, MVS/XA can initialize it. When initialization is complete, multiprocessing operations can proceed.

- Operation

  Normal operation proceeds with each processor receiving work from the MVS/XA dispatcher. The dispatcher normally gets control after a system event occurs or when a unit of work is complete. However, if one processor has entered the wait state because it had no work to perform, another processor can tell the idle processor that new work has arrived. This kind of communication is called **shoulder-tapping.** It is accomplished by a SIGP instruction that causes an external interruption in the addressed processor.

- Reconfiguration

  When the operator configures a processor offline or online, MVS/XA-initiated interprocessor communication is necessary. For example, if the master scheduler is running in processor A when a CONFIG command is received to configure processor B offline, processor A issues a SIGP instruction to tell processor B to stop. Processor B enters the stopped state just as it would had the STOP key on the processor's system console been pressed. To configure processor B back online, processor A issues a SIGP restart instruction to restart processor B just as though the RESTART key had been pressed.

- Recovery

  When a processor fails due to a software malfunction the machine check handler (MCH) issues a SIGP emergency signal (EMS) instruction to the other processors. The EMS causes an external interruption on the functioning processors. The first processor to receive the interruption initiates recovery processing for the failing processor. As part of recovery processing, the functioning processor might issue SIGP instructions to determine the status of the failing processor. If the status can be obtained, the MVS/XA recovery routines have a better chance of succeeding. These recovery routines, known as alternate CPU recovery (ACR) routines, are described in Chapter 10, "Recovering From Errors."

Hardware-Initiated Communication

In addition to the signals exchanged between processors through use of the SIGP instruction, the hardware supports direct communication between the processors. This communication is necessary to ensure:

- Clock Synchronization

  In a tightly-coupled MP configuration, there is more than one time-of-day (TOD) clock. (Note that the 3081, a dyadic tightly-coupled processor, has only one.) The TOD clocks must be synchronized when a tightly-coupled MP system is initialized or when a processor is reconfigured to be online.

- Storage Control

  Because storage is shared among the processors, the processors must communicate with each other to ensure that all references to shared storage refer to the most current data. Therefore, each processor (for example, processor A) indicates when it modifies the contents of a real storage location. Another processor (for example, processor B) can determine whether its high-speed buffer currently contains the contents of that same real storage location. If so, this copy of the storage is no longer current; processor B invalidates the copy in its buffer.

- Recovery

  When a processor experiences a failure that causes it to enter the check-stop state, the failing processor generates a malfunction-alert (MFA) interruption on the other processors, one of which then attempts recovery. Alternate CPU recovery (ACR) routines, described in Chapter 10, "Recovering From Errors," receive control and remove the failing processor from the configuration so that MVS/XA can continue running on an operative processor.

Chapter 5. Managing System Resources

An MVS/XA system, like other computer systems, has three broad categories of computer system resources: processors, real storage, and I/O devices. Managing system resources is the responsibility of the MVS/XA component, the **system resources manager** (SRM). SRM has two objectives:

- To achieve optimal use of the system resources from the system point of view (throughput)
- To achieve optimal use of system resources from the point of view of the individual address space (response and turnaround time)

This chapter describes how SRM attempts to meet these objectives, including the decisions it makes and the factors it considers in making those decisions. The installation can influence almost all of the decisions made by SRM routines by means of member IEAIPSxx, the installation performance specification (IPS), and member IEAOPTxx of the SYS1.PARMLIB data set.

SRM Decisions

SRM's two objectives are contradictory. Optimizing throughput implies keeping resources busy; meeting the installation's objectives for response and turnaround time (as reflected in the IPS) implies the availability of any resource when it's required. SRM makes decisions that represent trade-offs between its two conflicting objectives.

The decisions SRM makes include the following:

- Which address spaces should be permitted access to the system's resources (that is, swapped in)
- When to steal pages and which pages to steal
- When to change the dispatching priority of address spaces
- Which device should be allocated, when allocation routines have a choice of devices
- When to inhibit the creation of new address spaces

Functional Areas of SRM

To reach its decisions, SRM is divided into three major functional areas:

- **SRM control**, which determines the processing required and routes control to the appropriate SRM routines. SRM control decides when and which address spaces will be swapped in or out. To make this decision, it obtains recommendations from the other functional areas of SRM: the workload manager and the resource manager.
- **Workload manager**, which monitors the use of resources by the various address spaces. It gives swapping recommendations to SRM control. These recommendations attempt to maintain each address space's use of system resources as specified in the IPS.

- **Resource manager**, which monitors system-wide use of resources to determine if they are over-utilized or under-utilized. It makes swapping recommendations to SRM control that are intended to achieve a balance between throughput and response time. In addition, the resource manager is responsible for implementing other SRM controls related to the use of resources:

  - Inhibiting the creation of new address spaces or stealing pages when certain shortages of real storage exist

  - Changing the **dispatching priority** of address spaces, which controls the rate at which the address spaces are allowed to consume resources

  - Choosing the device to be allocated, if a choice of devices exists, in order to balance the use of I/O resources.

Communicating with SRM

Other system components communicate with SRM by means of the **SYSEVENT macro instruction**. All SYSEVENTs have a code, which indicates the processing SRM is to do. These codes fall into one of two categories:

- SYSEVENTs that notify SRM of a change in status for a particular address space or for the system as a whole. For example, a SYSEVENT is issued:

  - when real storage has been configured into or out of the system

  - when an address space is to be created (if a shortage of SQA or pageable storage exists, SRM will prohibit the creation of an address space)

  - when an address space has been deleted

  - when an address space enters a long wait (SRM will swap the address space out of real storage)

  - when an initiator selects or terminates a job

  - when a swap-in starts or a swap-out completes.

- SYSEVENTs that invoke SRM's decision-making functions. Such a SYSEVENT is issued:

  - when allocation routines can choose the devices allocated to a request (SRM will recommend one of the devices)

  - when a time interval expires. (The timer-interval SYSEVENT is the exclusive means to invoke most of SRM's algorithms, which provide data on which SRM bases its decisions.)

Most SYSEVENTs invoke SRM control which, in turn, calls the resource or workload manager. The remainder of this chapter describes in greater detail SRM control, the workload manager, and the resource manager.

SRM Control

SRM control is the dispatcher of SRM. It schedules actions and algorithms to be performed by other SRM routines and is responsible for the swapping of address spaces.

The installation provides guidelines for SRM's swap decisions by defining a **domain** for each distinct type of work (for example, batch work). For each domain, the installation defines a **minimum MPL** and **maximum MPL** (multiprogramming level) and the domain's importance relative to other domains. The MPLs state the minimum and maximum number of address spaces in each domain that should be in real storage (that is, swapped in) at the same time.

Within the boundaries of the minimum and maximum MPL and based on such factors as the total utilization of system resources, SRM periodically computes an optimal MPL for each domain, called the **target MPL.** The objective of the swap analysis performed by SRM control is to maintain the MPL of each domain at its target value.

Swap Analysis

Swap analysis is triggered by several events, such as when a user becomes ready to execute or when a time interval expires. The swap analysis must answer two questions: **whether** a swap is necessary; and, if so, **which** address space(s) to swap.

The are four types of swaps SRM considers necessary:

1. **Unilateral Swap-Out**

   If SRM locates any domain(s) whose current MPL exceeds its target, SRM control swaps out the required number of address spaces to lower the domain's MPL to its target value.

2. **ENQ Exchange**

   If a swapped-out address space is enqueued on a resource requested by another user, SRM control swaps in the enqueued user. Note: enqueuing is a technique for gaining control of a resource and is explained in Chapter 6, "Supervising the Execution of Work."

3. **Exchange Swap**

   If SRM determines that an exchange of a swapped-in address space and swapped-out address space will redress an imbalance in the use of resources, the exchange swap occurs.

4. **Unilateral Swap-In**

   IF SRM locates any domain(s) whose current MPL is less than the target, SRM control swaps in the required number of address spaces to raise the current MPL to its target value.

To determine which address space(s) within a domain to swap in or out, SRM control asks the workload manager and resource manager for swap recommendations, which take the form of **swap recommendation values (RVs).** The workload manager's RVs aim to maintain an address space's use of resources as

specified in the IPS. The resource manager's RVs aim to correct imbalances in I/O or processor utilization. By combining the RVs of the workload manager and resource manager, SRM control makes trade offs between its two objectives: distributing resources as specified in the IPS and optimizing throughput.

The Workload Manager

The workload manager has three basic functions:

- To monitor service rates - the rates at which system resources are being provided to individual address spaces
- To provide swapping recommendations requested by SRM control
- To collect data for certain measurement tools such as the Resource Measurement Facility (RMF)

The workload manager measures the rate at which resources are used in terms of **service units** per second. Service units are computed as a combination of three basic system resources: processor time used, I/O activity, and real storage frames occupied. The service rate is the result of dividing the number of service units by a time interval, which includes both the time an address space is swapped into real storage and the time it is swapped out but otherwise ready to execute.

To arrive at a swapping recommendation, the workload manager measures the service rates of different address spaces and compares them in light of factors defined by the installation in the IPS (installation performance specification). By means of these factors, the installation can instruct SRM to give certain users better service at the expense of other users. For example, assume two address spaces exist in real storage and one must be swapped out; the installation-defined IPS factors will dictate how the workload manager views measured service rates:

- Address space A has a higher service rate than address space B. Based on IPS factors associated with these two address spaces, the workload manager determines that address space B should be swapped out.
- Address space A has a lower service rate than address space B. A different IPS indicates that address space A is more important and, based on this, the workload manager determines that address space B should be swapped out.
- Address space A and address space B have identical service rates. Again, IPS factors indicate which address space is more important and which, therefore, should remain in storage.

The workload manager passes its swap recommendations to SRM control, which combines them with recommendations from the resource manager.

The Resource Manager

The resource manager employs algorithms that are concerned with improving the system-wide use of resources (as contrasted to an individual address space's use of resources, which is the concern of the workload manager). The resource manager's routines can be divided into four functional areas:

- Storage management, which is concerned with SRM's decisions to steal pages and to prevent the creation of new address spaces

- I/O management, which is concerned with SRM's swap decisions and device allocation decisions
- Processor management, which is concerned with SRM's swap decisions and decisions to change an address space's dispatching priority
- Resource monitoring, which is concerned with adjusting the target MPLs of individual domains based on the need to raise or lower the system-wide multiprogramming level

Storage Management

SRM's storage management routines take action when shortages of the following are detected: available frames in real storage; space in the system queue area (SQA) that causes the SQA to expand into the common service area (CSA); slots on auxiliary auxiliary storage; and pageable frames in real storage.

The system maintains an **available frame queue**, which indicates the number of available frames in real storage. When the number of available frames falls below a "low" threshold, SRM storage management routines begin to steal the least-recently used pages from the working sets of address spaces in real storage. The storage management routines continue stealing pages until the count of available frames plus the number of pages stolen exceeds an "OK" threshold for the available frame queue.

SQA shortages are detected by the virtual storage manager (VSM), which calls SRM's storage management routines when a shortage is detected. The storage management routines prevent the creation of new address spaces until the shortage is relieved. The routines also write messages to the operator when the shortage is detected and when the shortage is relieved.

SRM's storage management routines periodically verify that the number of available auxiliary storage slots has not fallen below a certain limit. Shortages of pageable real storage are detected by real storage management (RSM) when the percentage of fixed frames to total frames exceeds a certain limit; RSM then notifies SRM's storage management routines. The action taken by SRM for shortages of auxiliary storage slots or pageable real storage is the same; SRM:

- Prevents the creation of new address spaces
- Delays newly-initiated jobs
- Sets the multiprogramming level in each domain to its minimum MPL
- Swaps out the user who is acquiring slots at the greatest rate (for shortages of auxiliary storage) or the user who has the most fixed frames (for shortages of real storage)
- Notifies the operator of the shortage and the identity of the swapped-out user

When the shortage is relieved, creation of address spaces is again allowed, the operator is notified, and address spaces that were swapped out are again made eligible for swap-in.

I/O Management

SRM's I/O management routines are called to:

- Choose a device when allocation routines have a choice of devices to allocate (device allocation)
- Give swap recommendations to SRM control (I/O load balancing)

In both cases, the objective of I/O management is to balance I/O activity across channel paths and, thereby, make optimal use of the channel subsystem. SRM uses the concept of logical paths. A logical path is the set of physical channel paths leading to a single device. Any other devices that share the same physical paths, share the same logical path. Channel paths are described in Chapter 7, "Satisfying I/O Requests."

When choosing a device for allocation, the device allocation algorithm seeks candidates on the logical path that has the lowest utilization. For direct access devices, it chooses the device with the least delays in accessing allocated data sets. When it gives swap recommendations to SRM control, the I/O load balancing algorithm bases its recommendations on the extent to which the swap-in or swap-out of a user would correct a detected I/O imbalance: it recommends, via swap recommendation values, that a significant user of an over-used logical path be swapped out; or that a significant user of an under-used logical path be swapped in.

Processor Management

Processor management routines have three responsibilities:

- Controlling the APG (automatic priority group) subset of dispatching priorities
- Preventing the swap-out of users who are enqueued on resources required by other users
- Making swap recommendations to correct under-utilization or over-utilization of the processor

The **APG** is a range of dispatching priorities under the control of SRM. Dispatching priority controls the rate at which address spaces are allowed to consume resources after they have been given access to those resources. By placing jobs in the APG range, the installation, via the IPS and SRM, can alter the dispatching priorities of address spaces as their execution characteristics change. The APG is the primary means by which SRM controls nonswappable address spaces.

The APG has at least one group of dispatching priorities. Each group is divided into three categories: mean-time-to-wait (MTTW), rotate priority, and fixed priorities.

- The **MTTW** can be used to increase system throughput by increasing processor and I/O overlap (that is, the processor is not waiting while I/O requests are satisfied). Users in the MTTW group have a dispatching priority based on the user's mean execution time before entering a wait state; users who quickly release the processor receive a high priority within the MTTW group.

- The **rotate priority** can be used to ensure that one address space does not dominate the processor in relation to other address spaces also assigned the rotate priority. Processor management routines periodically reposition the address space that is highest in the rotate priority group to the bottom of the group.

- SRM does not change **fixed priorities**. They are available so that the installation can associate, via the IPS, a different fixed priority with different periods in the life of a job or transaction.

In the case of address spaces with users enqueued on resources in demand by other users, processor management routines prevent their swap-out until they have released the resource or executed for a fixed period of time (whichever occurs first). The installation can specify the execution time interval via an SRM tuning parameter.

If processor management routines determine that the processor is over- or under-utilized, they search for heavy processor users and calculate swap recommendation values for swap-out (to correct over-use) or swap-in (to correct under-use). A heavy processor user is one that meets or exceeds a certain mean execution time before entering the wait state. The processor is considered over-utilized if, during the period under consideration, it did not enter the wait state and any ready address space on the dispatching queue was not dispatched. The processor is considered under-utilized when its use is less than a certain percentage of total possible processor use. Processor management routines take into account the extent to which the processor is over- or under-utilized when computing swap recommendation values for SRM control.

Resource Monitoring

The resource monitoring function of the resource manager periodically checks several system resource use indicators such as processor use. If measured resource use (averaged over a number of sample intervals) is greater than a "high" threshold or less than a "low" threshold for that indicator, the resource monitoring function recommends that the system-wide multiprogramming level (MPL) be lowered or raised. (The system-wide MPL is the total number of address spaces in the system that are swapped in.)

If the system-wide MPL is to be raised or lowered, resource monitoring routines then identify the individual domain whose MPL will be raised or lowered to achieve the recommended system-wide MPL. The domain selected for MPL adjustment depends on the relative importance of the domains, as defined by the installation in the installation performance specification (IPS).

Chapter 6. Supervising the Execution of Work

The MVS/XA component known as the **supervisor** provides the controls needed for multiprogramming. The supervisor takes control once work is brought into real storage where it has access to the processor. This chapter describes the following supervisor controls:

- **Interruption processing**. In order to achieve multiprogramming, there must be some technique to switch control from one routine to another so that, for example, when routine A must wait for an I/O request to be satisfied, routine B can execute. In MVS/XA, this switch is achieved by **interruptions**, which are events that alter the sequence in which the processor executes instructions. When an interruption occurs, the hardware gives control to the supervisor which saves the execution status of the interrupted routine, analyzes the interruption, and passes control to the appropriate routine to process the interruption.

- **Creating dispatchable units of work**. The supervisor requires some way to identify and keep track of all the work in the system. It does this by representing each unit of work with a control block. Two types of control blocks represent dispatchable units of work in MVS/XA systems: **task control blocks (TCBs)**, which represent tasks executing within an address space; and **service request blocks (SRBs)**, which represent high priority system services.

- **Dispatching work**. After supervisor routines process interruptions, they either return control to the routine that was interrupted or pass control to a routine called the **dispatcher**. Which action occurs is described in detail in the topic "The Interruption Handlers." The dispatcher determines which unit of ready work, of all the ready units of work in the system, has the highest priority and passes control to that unit of work.

- **Serializing the use of resources**. In a multiprogramming system, almost any sequence of instructions can be interrupted, to be resumed later. If that set of instructions manipulates or modifies a resource (for example, a control block or a record in a data set), the supervisor must prevent other programs from using the resource until the interrupted program has completed its processing of the resource.

  In MVS/XA, the supervisor provides two techniques for serializing the use of resources: **enqueuing** (via the ENQ or, for shared DASD, RESERVE macro instruction) and **locking**. All users can issue ENQ or RESERVE, but only supervisor routines can use locking to serialize the use of resources.

Interruption Processing

An interruption is an event that alters the sequence in which the processor executes instructions. An interruption may be planned (specifically requested by the program the processor is currently executing) or unplanned (caused by an event that may or may not be related to the task currently executing). There are six types of interruptions:

- **SVC (supervisor call) interruptions**, which occur when the program issues an SVC instruction. An SVC is a request for a particular system service; for example, to open a data set (SVC 19 - OPEN), or to obtain storage (SVC 4 - GETMAIN), or to write a message to the operator (SVC 35 - WTO/WTOR).

- **I/O interruptions**, which occur when the channel subsystem signals a change of status. For example, an I/O operation completes, an error occurs, or a device becomes ready.

- **External interruptions**, which indicate any of several events. For example, a time interval expires, or the operator presses the interrupt key on the console, or one processor receives a signal from another processor.

- **Restart interruptions**, which occur when the operator selects the restart function at the console or when a restart SIGP (signal processor) instruction is received from another processor.

- **Program interruptions**, which are caused by program errors (for example, the program attempts an invalid operation), page faults (the program references a page that is not in real storage), or requests to monitor an event.

- **Machine check interruptions**, which are caused by machine malfunctions.

The supervisor includes six **first level interruption handler** (FLIH) routines to process the six types of interruptions: an SVC FLIH, I/O FLIH, external FLIH, restart FLIH, program FLIH, and machine check FLIH. When an interruption occurs, the hardware saves the key information about the program that was interrupted and, if possible, disables the processor for further interrupts of the same type. It then routes control to the appropriate first level interruption handler routine. The PSW is a key resource in this process.

The Program Status Word

The program status word (PSW) controls the order in which instructions are executed and indicates the status of the system in relation to the program currently being executed. Even though each processor has only one PSW, it is useful to think of three types of PSWs in order to understand interruption processing. The three PSWs are: the current PSW, new PSWs, and old PSWs.

The **current PSW** is the hardware location in the processor that indicates the next instruction to be executed. It also indicates whether the processor is **enabled** or **disabled** for I/O interruptions, external interruptions, machine check interruptions, and certain program interruptions. When the processor is enabled, these interruptions can occur. When the processor is disabled, these interruptions are ignored or remain pending. A pending interruption is processed when the unit of work that is executing in the disabled state completes. (The processor is never disabled for SVC, restart, certain program interruptions, and certain machine checks.)

There is a **new PSW** and an **old PSW** associated with each of the six types of interruptions. The new PSW contains the address of the first level interruption handler routine that can process its associated interruption. If the processor is enabled for interrupts when an interruption occurs, the MVS/XA hardware switches PSWs by:

1. Storing the current PSW in the old PSW associated with the type of interruption that occurred

2. Moving the contents of the new PSW for the type of interruption that occurred into the current PSW

The current PSW, which indicates the next instruction to be executed, now contains the address of the appropriate FLIH routine to handle the interruption (see Figure 6-1); this switch has the effect of transferring control to the appropriate first level interruption handling routine.

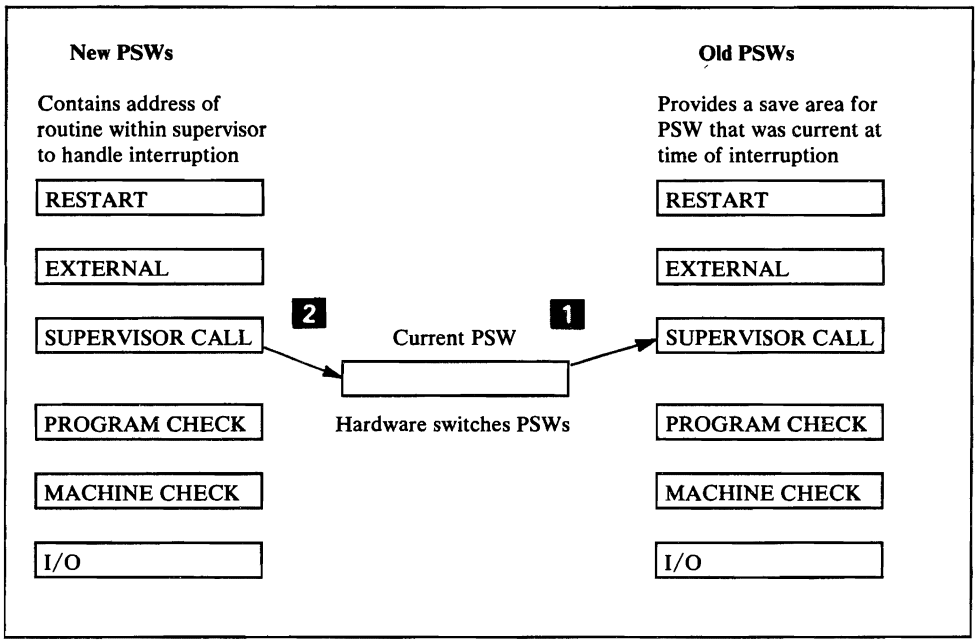


Figure 6-1. The Use of Program Status Words (PSWs) in Interruption Processing

The Interruption Handlers

The first level interruption handler (FLIH) that receives control saves the status (general registers and the old PSW) of the unit of work that was interrupted, analyzes the interruption, enables the processor for further interruptions, and determines the control program action required. Specifically:

- The **SVC first level interruption handler** determines the type and location of the requested SVC routine and, if the requested routine requires that the caller be authorized, checks that the caller has the appropriate authorization. (The request is denied if the caller lacks the required authorization.) There are several types of SVC routines, and each type has different execution characteristics. For example, some types of SVC routines reside in the nucleus, others in the link pack area; some types can issue other SVCs, other types cannot. If the requested SVC routine is a type that can issue other SVCs, the SVC FLIH builds a control block called an SVC request block (SVRB) for the requested routine. The SVRB is needed to save status information about the routine so that it can be resumed after the additional SVC interruption has been processed. After checking for proper authorization and, if necessary, building an SVRB, the SVC FLIH passes control to the requested SVC routine.

- The **I/O first level interruption handler** passes control to the input/output supervisor (IOS). IOS performs all processing for I/O requests and controls all I/O error processing.

- The **external first level interruption handler** determines the cause of the external interruption and passes control to the appropriate external service routine.

- The **restart first level interruption handler** routes control to the recovery termination manager (RTM). RTM is the focus of Chapter 10, "Recovering From Errors."

- The **machine check first level interruption handler** records all machine checks (hardware problems) and, if the machine check cannot be corrected by hardware, calls the recovery termination manager (RTM).

- The **program first level interruption handler** determines the cause of the program interruption and, depending on the cause, passes control to one of the following:

  - Real storage manager (RSM), if the program interruption was caused by a page or segment fault. RSM determines if the fault is valid and, if it is, starts the processing necessary to either build the page table or bring the referenced page into real storage.

  - System trace (TRACE), if the program interruption was a trace table exception. This indicates a full buffer condition, which system trace handles before using another buffer. System trace is described in Chapter 9, "Monitoring System Activity,"

  - Generalized trace facility (GTF), if the interruption occurred as the result of a request to monitor a class of events. GTF, also described in Chapter 9, "Monitoring System Activity," records the event.

  - Serviceability level indication processing (SLIP) if the interruption occurred as a result of a request to monitor an instruction fetch, successful branch, or storage alteration event. SLIP, described in Chapter 9, "Monitoring System Activity," performs a diagnostic action for such an event.

  - A user-provided program-interruption exit routine, if the program interruption was caused by an error in user code such as using an incorrect address or attempting to execute privileged instructions, and the user issued a specify program interruption element (SPIE) macro instruction to provide an error-handling routine.

  - The recovery termination manager (RTM), if the program interruption was caused by an error in system code or in user code that does not also include SPIE.

The routine that receives control after the interruption is processed depends on whether the interrupted unit of work was non-preemptive. A non-preemptive unit of work can be interrupted but must receive control after the interruption is processed. All SRBs are non-preemptive; a TCB is non-preemptive if it is executing a non-preemptive SVC (the installation identifies which SVCs will be non-preemptive during system generation). If the interrupted unit of work was preemptive, the dispatcher receives control and determines which unit of work should be performed next.

Figure 6-2 summarizes the processing of interruptions; for more information on the dispatcher, see "Dispatching Work."

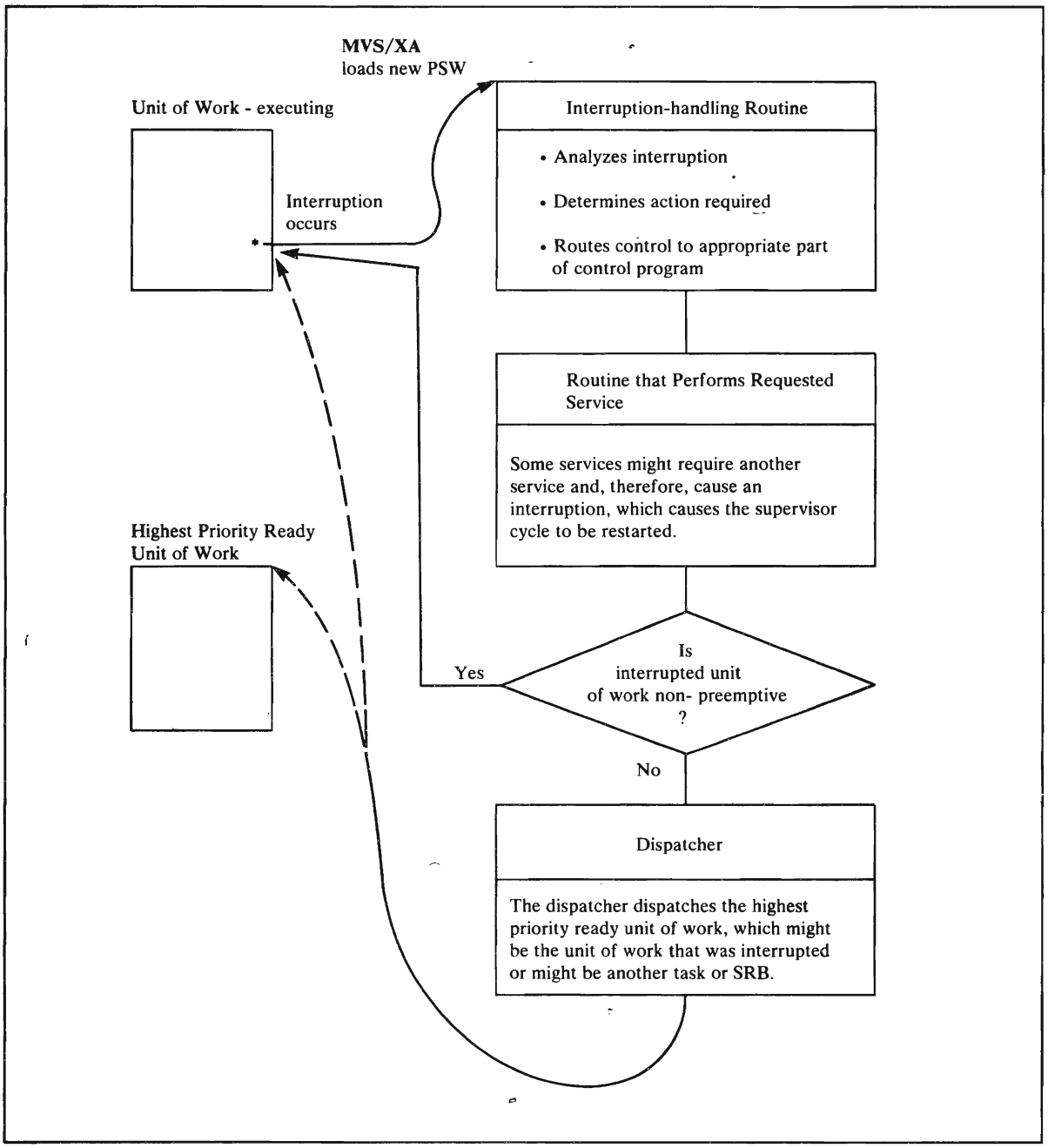


Figure 6-2. Summary of Interruption Processing

Creating Dispatchable Units of Work

In MVS/XA, dispatchable units of work are represented by two different control blocks:

- Task control blocks (TCBs), which represent tasks executing within an address space, such as user programs and system programs executed to support the user programs.
- Service request blocks (SRBs), which represent requests to execute a service routine. SRBs are typically created when one address space detects an event

that affects a different address space; they provide one mechanism for communication between address spaces.

Task Control Blocks (TCBs)

Task control blocks (TCBs) are created in response to an ATTACH macro instruction. By issuing ATTACH, a user or system routine causes the supervisor to begin the execution of the program specified on the ATTACH macro as a subtask of the caller's task. As a subtask, the specified program can compete for processor time and may use certain resources already allocated to the caller's task.

The ATTACH macro instruction causes an SVC interruption. The SVC interruption handler branches to the ATTACH SVC routine to perform the requested service. The ATTACH routine does the following:

- Obtains storage for a new TCB
- Places in the new TCB information needed to control the subtask
- Places the new TCB on the chain of TCBs for that address space
- Branches to program management routines to locate the first program to be executed for the new subtask and, if necessary, fetches the program from a program library.

The region control task (RCT), which is responsible for preparing an address space for swap-in and swap-out, is the highest priority task in an address space. All tasks within an address space are subtasks of the RCT. The RCT's TCB is pointed to from the address space control block extension (ASXB) and points to the next TCB in the address space. Figure 6-3 illustrates the address space dispatching queue of the TCBs for batch jobs, operator-started jobs, and TSO users.

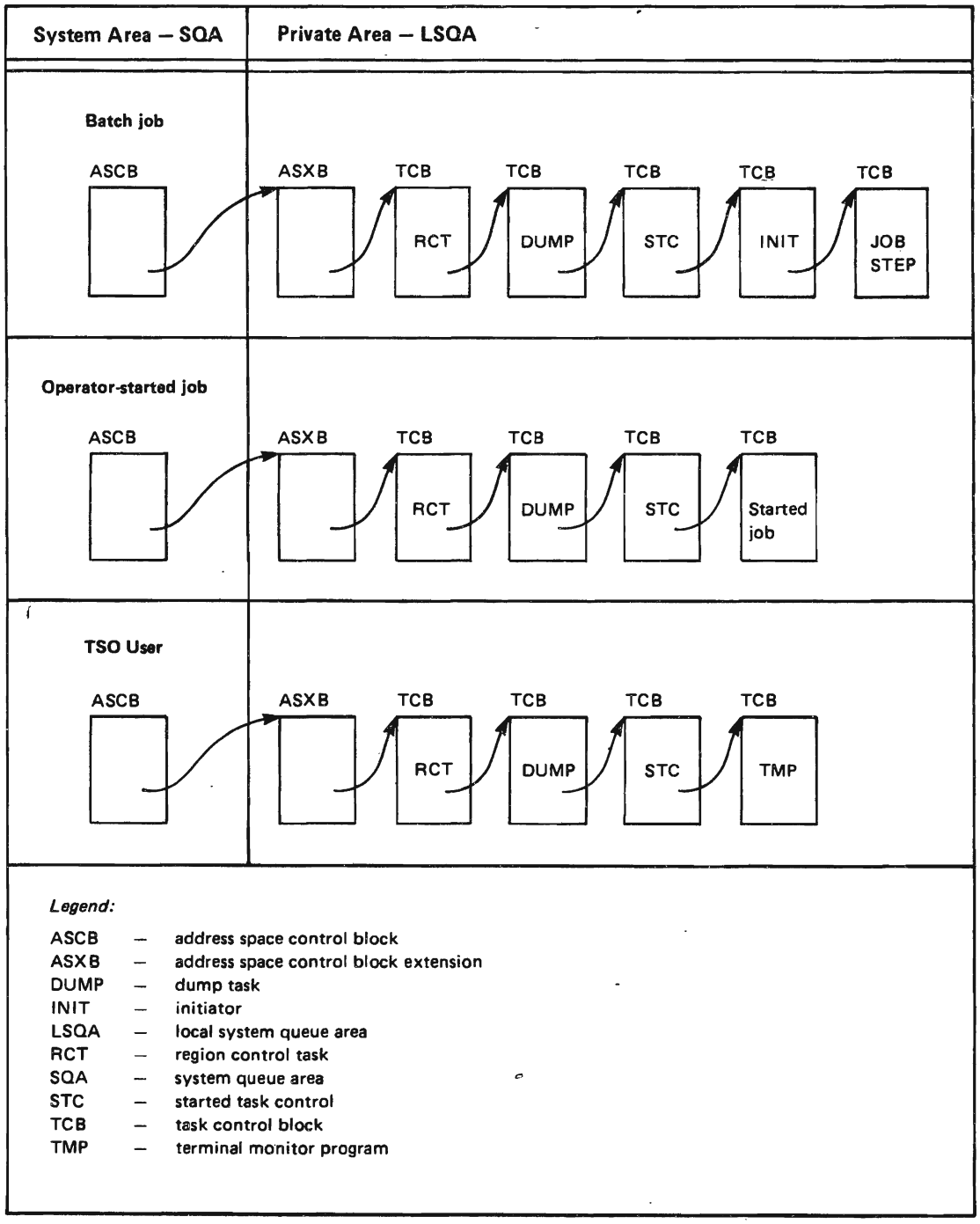


Figure 6-3. Address Space Task Control Block (TCB) Dispatching Queues

Service Request Blocks (SRBs)

An SRB represents a request to perform a service in a specified address space. Typically, an SRB is created when one address space is executing and an event occurs that affects a different address space.

Only supervisor state, key 0 functions create an SRB. They obtain storage and initialize the control block with such things as the identity of the target address

space and pointers to the code that will process the request. The component creating the SRB then issues the SCHEDULE macro and indicates whether the SRB has global (system wide) or local (address space wide) priority. SCHEDULE places the SRB on the appropriate dispatching queue where it will remain until it becomes the highest priority task on the queue.

SRBs with a global priority have a higher priority than that of any address space, regardless of the actual address space in which they will be executed. SRBs with a local priority have a priority equal to that of the address space in which they will be executed, but higher than that of any TCB within that address space. The assignment of global or local priority depends on the "importance" of the request; for example, SRBs for I/O interruptions are scheduled at a global priority, to minimize I/O delays.

SRBs are non-preemptive. Thus, if a routine represented by an SRB is interrupted, it will receive control after the interruption has been processed. In contrast, a routine represented by a TCB is preemptive. If it is interrupted, control returns to the dispatcher when the interruption handling completes. The dispatcher then determines what task, of all the ready tasks, executes next. Any, TCB, except one representing a task issuing a non-preemptive SVC, can be preempted.

An SRB can execute concurrently and in a different address space from the task that created it and issued the SCHEDULE macro. This means, among other things, that an SRB provides the means for asynchronous inter-address space communication. Such communication improves the availability of resources in a multiprocessing environment.

As an example, consider that, when address space A is executing, an I/O interruption occurs because an I/O operation requested by address space B has completed. The I/O interruption handler collects the necessary information about the interruption and creates and schedules the SRB to control the final processing of the completed I/O operation. The I/O interruption handler then starts any other I/O requests waiting for the I/O path used by the just-completed request and can accept any additional pending interruptions. Building the SRB allows faster re-use of the I/O path and less time when the processor is disabled for interruptions.

The SRB identifies the routine to process the completed I/O request and the address space in which the routine should execute. In the preceding example, the SRB would be executed in address space B, because that address space had requested the I/O operation.

Dispatching Work

Dispatching work consists of routing control to the highest priority unit of work that is ready to execute. The dispatcher, a supervisor routine, dispatches work in the following order:

1. Special exits. These are exits to routines that have a high priority because of specific conditions in the system. For example, if one processor in a tightly-coupled multiprocessing system fails, alternate CPU recovery (ACR) will be invoked by means of a special exit to recover work that was being executed on the failing processor.

2. SRBs that have global priority. If a global SRB cannot be dispatched (for example, the address space in which it will execute is swapped out), the dispatcher reschedules it at a local priority.

3. Ready address spaces in order of priority. An address space is ready to execute if it is swapped in and not waiting for some event to complete; an address space's priority is determined by the dispatching priority specified by the user or the installation. The address space control block (ASCB) contains the address space's dispatching priority; ASCBs that represent ready address spaces are queued in storage according to their dispatching priority. To select an address space, the dispatcher selects the first ready ASCB on the chain of ASCBs.

   After selecting the highest-priority ASCB, the dispatcher first dispatches SRBs with a local priority that are scheduled for that address space and then TCBs in that address space.

If there is no ready work in the system, the dispatcher loads an enabled wait PSW.

The dispatcher receives control after a task is interrupted or becomes non-dispatchable, after an SRB completes or is suspended, (that is, an SRB is delayed because a required resource is not available), and from other supervisor routines that want higher priority work dispatched without waiting for an interruption to occur. The dispatcher saves the status of the unit of work relinquishing control, selects a unit of work, builds a program status word (PSW) for the selected unit of work, and issues a load PSW (LPSW) instruction, which causes the selected routine to receive control. That routine executes until an interruption occurs or until the routine voluntarily gives up control (for example, by issuing a WAIT SVC).

Serializing the Use of Resources

The supervisor provides two techniques for serializing the use of resources: enqueuing and locking. The primary function of these techniques is to provide orderly access to system resources needed by more than one user in a multiprogramming or multiprocessing environment.

To protect system resources from unauthorized users, IBM makes available the Resource Access Control Facility (RACF). RACF controls access by permitting only authorized users to perform authorized actions on protected resources. These resources include DASD data sets, DASD and tape volumes, display terminals, system and user programs, and application program transactions.

Enqueuing

Enqueuing is accomplished by means of the ENQ (enqueue) and DEQ (dequeue) macro instructions, which can be used by both user and system programs; or, for devices shared between systems, by means of the RESERVE and DEQ macro instructions. On ENQ or RESERVE, a user specifies the name(s) of one or more resources and requests shared or exclusive control of those resources. If the resources are to be modified, the user must request exclusive control; if the resources are not to be modified, the user should request shared control, which allows the resource to be shared by other users that do not require exclusive control. If the resource is not available, the requestor is suspended until it becomes available. The DEQ macro instruction is used to release control of a resource.

Global Resource Serialization

The global resource serialization component of MVS/XA processes the requests for resources that programs issue. It serializes access to resources to protect their integrity. An installation can connect two or more systems with channel-to-channel (CTC) adapters to form a global resource serialization complex to serialize access to resources shared among the systems in the complex. Chapter 8, "Entering and Scheduling Work"illustrates the use of global resource serialization in a multiprocessing configuration.

In a complex without global resource serialization, a RESERVE macro instruction applies to the entire shared DASD volume on which the resource resides; no other system can access any resource on the volume. With a global resource serialization complex, an installation can improve the availability of resources on shared DASD volumes by converting RESERVEs to apply to only the requested resource; other systems can then access other resources on the volume.

Locking

Locking serializes the use of system resources by supervisor routines and, in a tightly-coupled multiprocessing system, by processors. A lock is simply a named field in storage that indicates whether a resource is being used and who is using it. In MVS/XA, there are two kinds of locks: **global locks**, for resources related to more than one address space, and **local locks**, for resources assigned to a particular address space. Global locks are provided for non-reusable or non-shareable routines and various resources.

To use a resource protected by a lock, a routine must first request the lock for that resource. A part of the supervisor called the lock manager acquires and maintains all locks. If the lock is unavailable (that is, already held by a different program or processor), the action taken by the program or processor that requested the lock depends on whether the lock is a spin lock or a suspend lock:

- If a **spin lock** is unavailable, the requesting processor continues testing the lock until the other processor releases it. As soon as the lock is released, the requesting processor can obtain the lock and, thus, control of the protected resource. All of the global locks except the cross-memory-services locks are spin locks.

- If a **suspend lock** is unavailable, the unit of work requesting the lock is delayed until the lock is available; other work is dispatched on the requesting processor. The cross-memory-services global locks and all local locks are suspend locks.

Lock Hierarchy

A deadlock is the situation where two users request locks held by each other and simultaneously wait for the other to release its lock first. The result is a stalemate.

To avoid deadlocks, locks are arranged in a hierarchy, and a processor or routine can unconditionally request only locks higher in the hierarchy than locks it currently holds. For example, a deadlock could occur if processor 1 held lock A and required lock B; and processor 2 held lock B and required lock A. This situation cannot occur because locks have to be acquired in hierarchical sequence. Assume, in the preceding example, that lock A precedes lock B in the hierarchy. Processor 2, then, cannot unconditionally request lock A while holding lock B. It must, instead, release lock B, request lock A, and then request lock B. Because of

the hierarchy, a deadlock cannot occur. Figure 6-4 identifies the locks MVS/XA provides and lists them in hierarchical order.

Lock Name	Category	Type	Description (See note 1.)
RSMGL	Global	Spin	- Real storage management global lock - serializes RSM global resources.
VSMFIX	Global	Spin	- Virtual storage management fixed subpools lock - serializes VSM global queues.
ASM	Global	Spin	- Auxiliary storage management lock - serializes ASM resources on an address space level.
ASMGL	Global	Spin	- Auxiliary storage management global lock - serializes ASM resources on a global level.
RSMST	Global	Spin	- Real storage management steal lock - serializes RSM control blocks on an address space level when it is not known which address space locks are currently held.
RSMCM	Global	Spin	- Real storage management common lock - serializes RSM common area resources (such as page table entries).
RSMXM	Global	Spin	- Real storage management cross memory lock - serializes RSM control blocks on an address space level when serialization is needed to a second address space.
RSMAD	Global	Spin	- Real storage management address space lock - serializes RSM control blocks on an address space level.
RSM	Global	Spin	- Real storage management lock (shared/exclusive) - serializes RSM functions and resources on a global level.
VSMPAG	Global	Spin	- Virtual storage management pageable subpools lock - serializes the VSM work area for VSM pageable subpools.
DISP	Global	Spin	- Global dispatcher lock - serializes the ASVT and the ASCB dispatching queue.
SALLOC	Global	Spin	- Space allocation lock - serializes receiving routines that enable a processor for an emergency signal or malfunction alert.
IOSYNCH	Global	Spin	- I/O supervisor synchronization lock - serializes, using a table of lockwords, IOS resources.
IOSUCB	Global	Spin	- I/O supervisor unit control block lock - serializes access and updates to the UCBs. There is one IOSUCB lock per UCB.
SRM	Global	Spin	- System resources management lock - serializes SRM control blocks and associated data.
TRACE	Global	Spin	- Trace lock (shared/exclusive) - serializes the system trace buffer.
CPU	Global	Spin	- Processor lock - provides system-recognized (legal) disablement. (See note 2.)
CMSSMF	Global	Suspend	- System management facilities cross memory services lock - serializes SMF functions and control blocks. (See note 3.)
CMSEQDQ	Global	Suspend	- ENQ/DEQ cross memory services lock - serializes ENQ/DEQ functions and control blocks. (See note 3.)
CMS	Global	Suspend	- General cross memory services lock - serializes on more than one address space where this serialization is not provided by one or more of the other global locks. The CMS lock provides global serialization when enablement is required (See note 3.)
CML	Local	Suspend	- Local storage lock - serializes functions and storage within an address space other than the home address space. There is one CML lock per address space. (See note 4.)
LOCAL	Local	Suspend	- Local storage lock - serializes functions and storage within a local address space. There is one LOCAL lock per address space. (See note 4.)

Notes:

1. All locks are listed hierarchical order, with RSMGL being the highest lock in the hierarchy. (See also notes 2, 3, and 4.)
2. The CPU lock has no hierarchy in respect to the other spin type locks. However, once obtained, no suspend locks can be obtained.
3. The cross memory services locks (CMSSMF CMSEQDQ, and CMS) are equal to each other in the hierarchy.
4. The CML and LOCAL locks are equal to each other in the hierarchy.

Figure 6-4. Definition and Hierarchy of Locks

Chapter 7. Satisfying I/O Requests

An input/output (I/O) operation involves the movement of data between main storage and an I/O device. **Input** is the movement of data from the device to main storage. **Output** is the movement of data in the reverse direction: from storage to the device. The I/O device may be a tape, a disk, printer, or a telecommunication device (such as a local display terminal or telecommunication control unit).

An MVS/XA system configuration can include more than 4000 I/O devices and can run many programs concurrently. To manage the I/O operations that these programs request, MVS/XA works with a separate processor dedicated to handling I/O operations. The I/O processor and its components are the **channel subsystem**.

MVS/XA initiates an I/O operation by signaling the channel subsystem. The channel subsystem, executing independently of the central processor, moves data between storage and the I/O device. The subsystem's ability to execute independently of the processor allows an I/O operation to overlap with central processor activity. This overlap is particularly important because an I/O operation takes a long time to complete compared to the time the central processor requires to execute a series of instructions. The overlap of I/O operations with processor activity is then one of the key ways that MVS/XA achieves efficient use of both the central processor and the computer system's I/O resources. Figure 7-1 illustrates the components involved in an I/O operation.

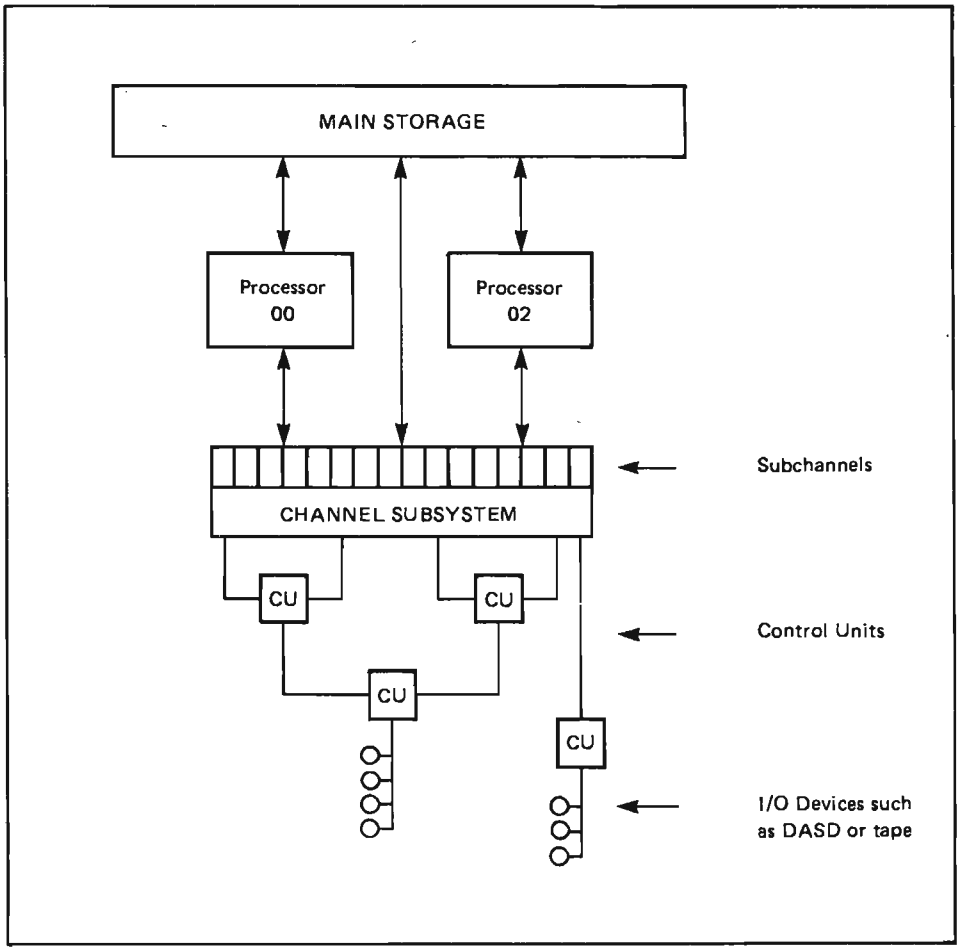


Figure 7-1. Components of the I/O Request

How I/O Data Moves Through the System

An I/O path can either be a conventional direct line I/O path or a telecommunication I/O path. The **conventional I/O path** consists of main storage, the channel subsystem, a control unit, and I/O devices such as disks and tapes used for local long-term storage of computer users' data and programs. Output data moves from storage to the device. Input data moves from the device to main storage. Figure 7-2 illustrates conventional I/O.

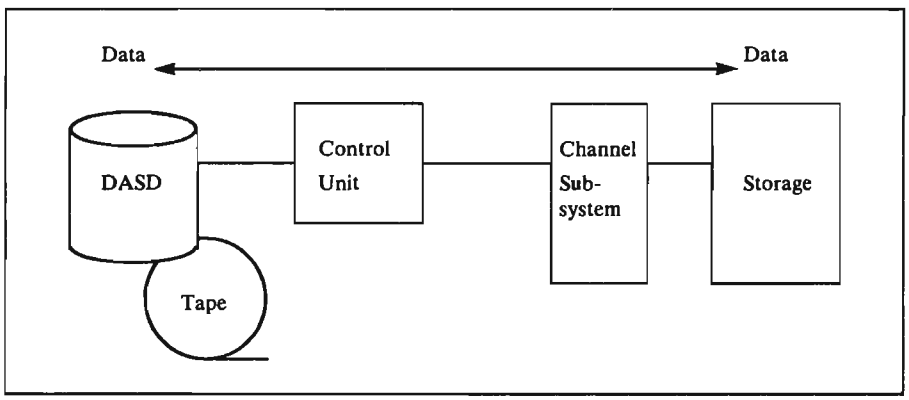


Figure 7-2. Conventional Input/Output

The **telecommunication I/O path** consists of storage, the channel subsystem, a communications controller, a data link, a control unit, and a device (usually a terminal). Input data moves from the terminal to the control unit to the data link. In the data link, the data is changed by a modem (modulator/demodulator) into a form that is transmitted over the communication line (such as a telephone line) to the processor location. At the processor location, another modem receives the data and converts it back to its original form. The data then moves through the communications controller and the channel subsystem to storage. Figure 7-3 illustrates telecommunication I/O.

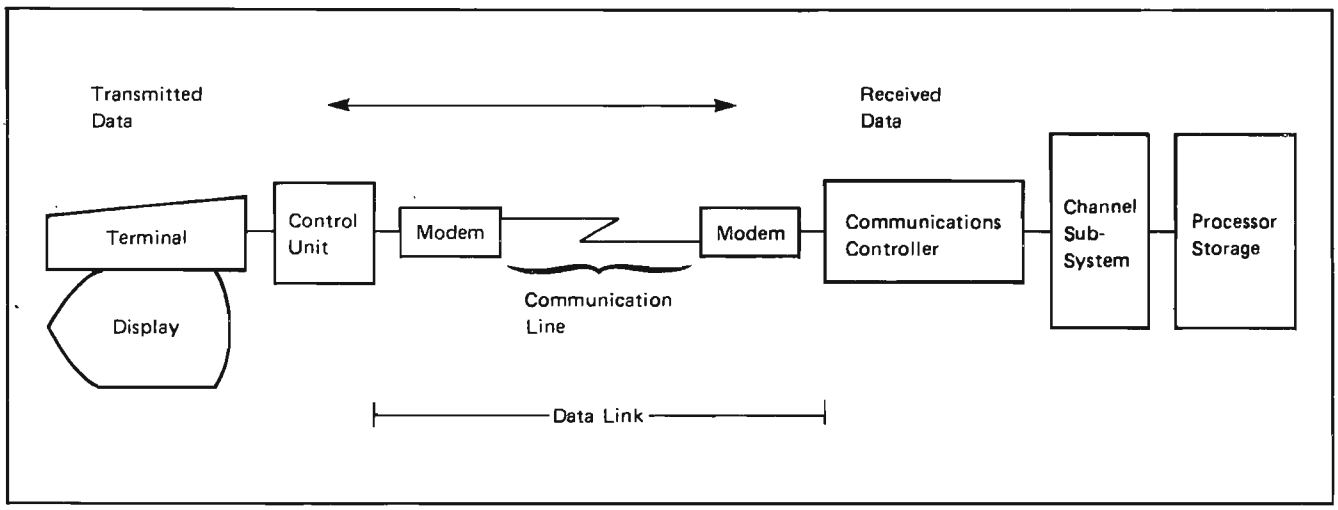


Figure 7-3. Telecommunication Input/Output

Output data uses the same path in reverse order; it moves from storage through the channel subsystem to the communications controller. From the communications controller, the data moves to the data link. In the data link, the data is changed by a modem into a form that is transmitted over a communication line to the terminal location. At the terminal location, another modem receives the data and converts it back to its original form. The terminal at the remote location then receives the data. Telecommunication I/O paths are used in an interactive computing environment where terminal users work with applications (such as TSO and IMS) that are executing on a processor at another location.

The I/O paths in the preceding figures illustrate, in a general way, the route data travels within an MVS/XA system. For the channel subsystem, the term **channel path** refers to a precise path of data transmission among specific components. For either conventional or telecommunication transfer, MVS/XA allows the definition of multiple I/O paths to a single device. That is, there can be more than one physical channel path to a specific device. Multiple paths enable the channel subsystem to schedule I/O requests to balance the load over physical channel paths and devices and also to allow continued access to a device if one of the multiple paths is inoperative.

Figure 7-4 shows the use of multiple channel paths to devices. Data can move between storage and disk A, disk B, and the tape device by using the path over channel path 01 or the path over channel path 02. If an input operation is under way from disk A through channel path 01, then channel path 02 can be used for an input operation from disk B or the tape device without having to wait for the input operation on disk A to complete. Data can move between storage and the communications controller (and subsequently to terminals C, D, and E by way of

the data link) by using the path over channel path 01 or the path over channel path 03. If terminal C and terminal D are using channel path 03 to interact with an application, terminal E can use another application and channel path 01 without affecting the response time of terminal C and terminal D.

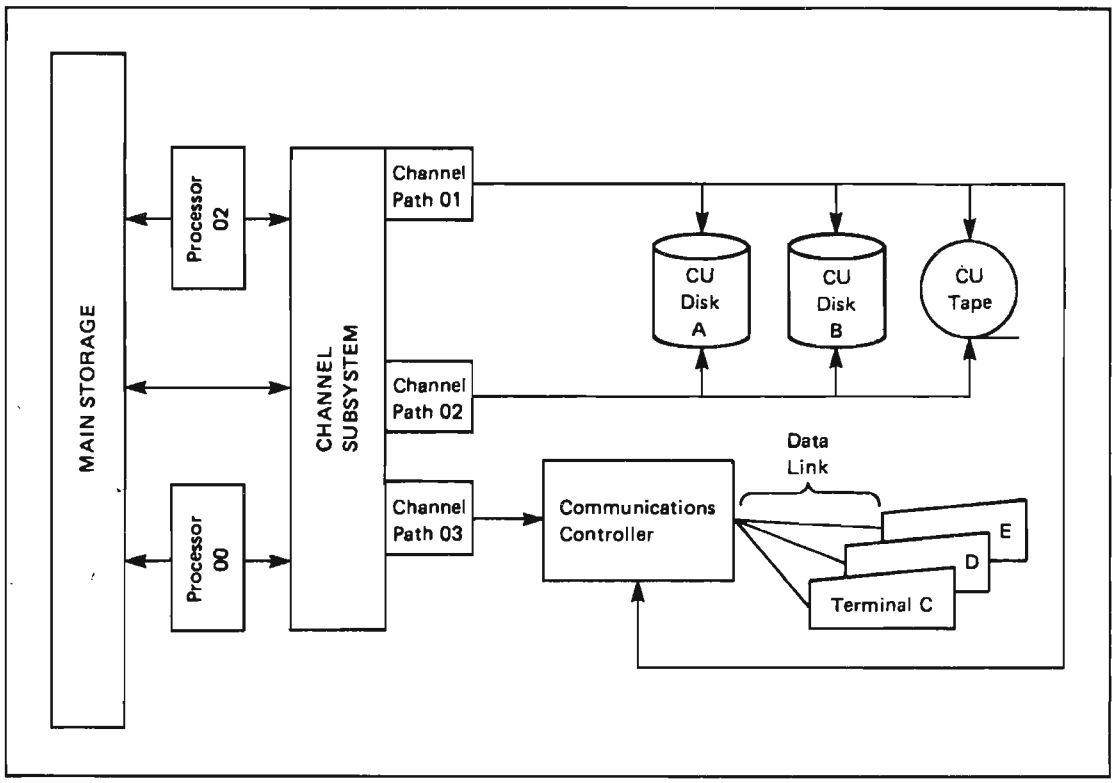


Figure 7-4. Multiple I/O Channel Paths

Controlling the I/O processing for jobs where multiple paths to an I/O device are available is a complex process. To manage this process the channel subsystem views groups of up to four channel paths and up to four physical control units as **logical control units**. The channel subsystem controls I/O processing (not only for one job, but for the many jobs that run concurrently in the system) by polling the possible channel paths to a device and assigning an available one to the next I/O request on the queue for the logical control unit. Figure 7-5 illustrates the relationship among logical control units, physical control units and channel paths.

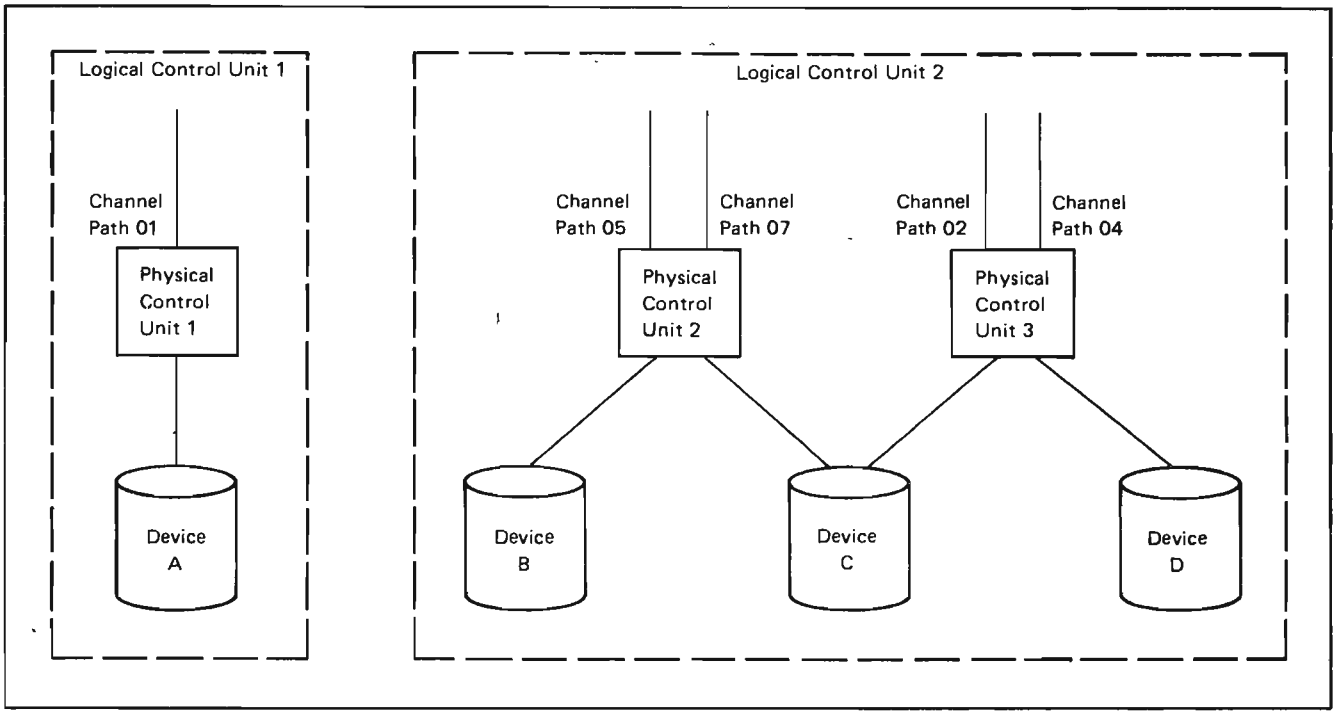


Figure 7-5. Logical Control Units

The channel subsystem identifies each device by a unique subchannel ID number. Usually, MVS/XA installations assign their own device number to each of the I/O devices. The MVS/XA component that initiates I/O requests, I/O supervisor (IOS), relates these two numbers to ensure that each I/O request is directed to the proper device. If that device is active with a previous IOS request when the current request is made, IOS holds the current request on a queue for that device.

The channel subsystem makes it possible for any one central processor in a multiprocessor system to access any of the I/O devices in the computer system. Each processor communicates with the channel subsystem, and the channel subsystem communicates with all of the I/O devices. The channel subsystem also protects against I/O delays and bottlenecks in the event of the failure of one processor in a multiprocessing system.

How an I/O Request Moves Through MVS/XA

MVS/XA is a flexible operating system that includes services that allow programmers to ignore the many details of I/O operations or to bypass or add to some phases of the I/O operations. Figure 7-6 illustrates the MVS/XA I/O services - access methods, an IOS driver, and IOS - and shows how they relate to one another when an I/O request is made by a user program. The discussion that follows describes how these services function in the typical situation where a programmer makes an I/O request by means of an access method that uses the EXCP processor as an IOS driver.

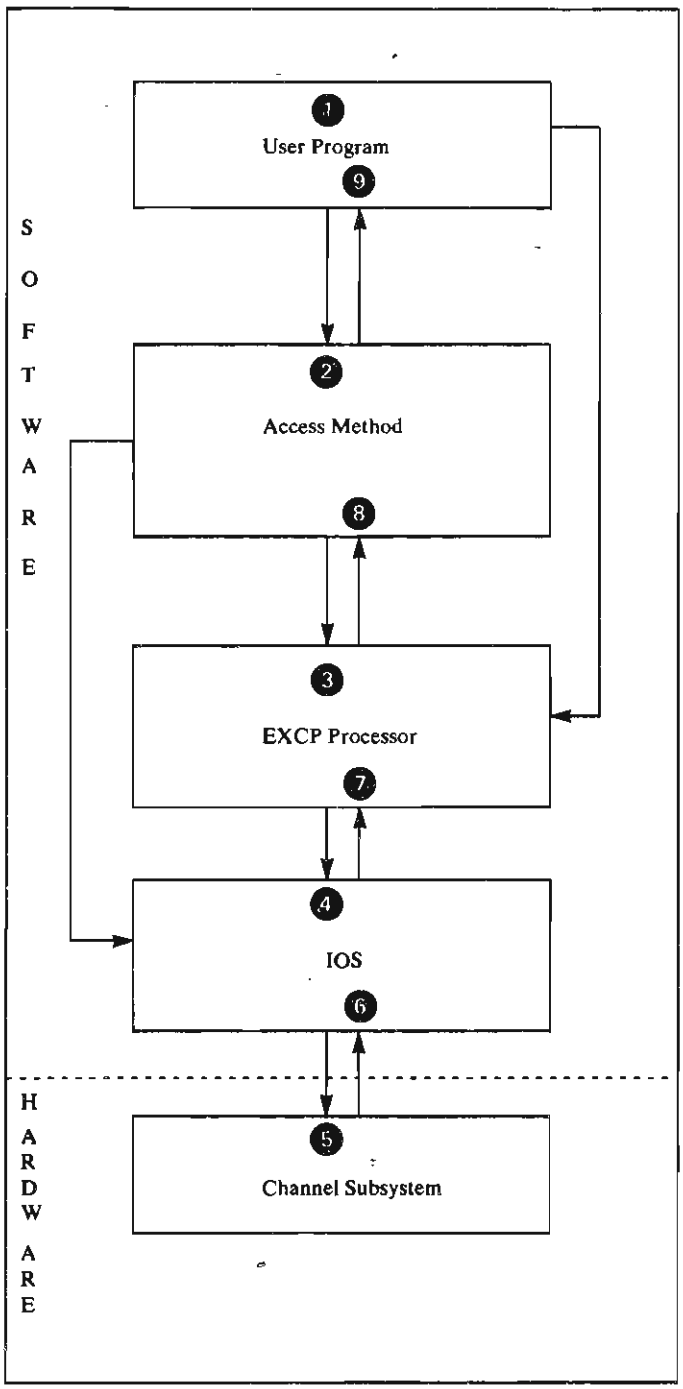


Figure 7-6. MVS/XA I/O Services

(1) The user program begins an I/O operation by issuing an OPEN macro instruction and asking for either input or output of data using an I/O macro instruction like, GET, PUT, READ, or WRITE, and specifying a target I/O device. An I/O macro instruction invokes an **access method** that interprets the I/O request and determines which system resources are needed to satisfy the request. The user program could bypass the access method, but it would then need to consider the many details of the I/O operation, such as the transmission characteristics of the path over which the data is to move, and the order in which to move the data between the I/O device and storage. The program would also have to create a

channel program, composed of instructions for the channel subsystem, and invoke the EXCP processor, an IOS driver, to handle the next phase of the I/O process.

(2) There are several MVS/XA access methods, each of which offers differing functions to the user program. These access methods fall into three categories: conventional access methods, telecommunication access methods, and the virtual storage access method (VSAM).

Conventional access methods move data between storage and I/O devices such as disks or tape; the program uses the I/O device to hold data the program would not normally keep in main storage. Telecommunication access methods move data over telecommunication I/O paths between storage and I/O devices such as display terminals; the I/O device is normally used to communicate and interact with the program rather than to hold data. The selection of an access method in either category (conventional or telecommunication) depends on how the data is currently organized and how the program plans to access it (randomly or sequentially, for example). VSAM is particularly designed for use with virtual storage. The final section of this chapter describes the various access methods in detail.

(3) To request the movement of data, either the access method or the user program presents information about the operation to the EXCP processor by issuing the EXCP macro instruction. EXCP translates this information into a format understandable to the channel subsystem and invokes the I/O supervisor (IOS).

(4) IOS places the request for I/O on the queue for the chosen I/O device, if necessary, and initiates the channel subsystem. Then, the central processor usually does other work until the channel subsystem indicates that the I/O operation has completed.

(5) The channel subsystem selects the best channel path for data transmission between storage and the device and controls the movement of data. When I/O is complete, the channel subsystem signals the completion by causing an **I/O interruption** and indicating exactly what occurred during the I/O operation.

(6) IOS evaluates the interruption, and returns control to EXCP.

(7) EXCP indicates that I/O is complete by posting the **event control block (ECB)** (created by the access method) and calling the dispatcher.

(8) When appropriate, the dispatcher re-activates the access method.

(9) The access method returns control to the user program which can then continue its processing.

A Closer Look at How an I/O Request Moves Through MVS/XA

As MVS/XA processes an I/O request, several software components communicate using MVS/XA macro instructions and programming conventions. Both the software and the hardware rely upon information stored in control blocks as the I/O process progresses. The following sections describe the role of each component involved in an I/O request and the control blocks and instructions they use.

The user program that issues the I/O request must describe the data set to be used and the specific operation to be performed on the data set. It supplies this information in a DD statement in the program's JCL and in a data control block (DCB), which the program creates. When the program issues an OPEN macro instruction, the DCB is filled in with all the relevant data set information.

OPEN Processing

When the user program issues an OPEN macro instruction, it invokes the system OPEN routines. These routines merge information from various sources to build a complete description of the data set. The information used comes from:

- The job file control block (JFCB), which contains data set and device information from the DD statement included in the JCL for the user program. After the device for the the data set has been allocated, the task I/O table (TIOT) entry points to the unit control block (UCB) for the required device and to the JFCB.

- The data set control block (DSCB) that describes the data set in great detail. It indicates, for example, how the data is organized, whether it it password protected, and when it was last referenced. For data sets on a direct access device, for example, the DSCB comes from the volume table of contents (VTOC) for the volume containing the data set.

- The data control block (DCB) built by the user's program. When OPEN processing begins, the DCB contains information about the data set organization and location that can be augmented by information from the JCL for the current job step.

  When OPEN processing is complete, the DCB contains all of the information about the data set merged during OPEN processing. This information includes the address of the access method routines which usually perform I/O operations. VSAM, and some subsystems use an equivalent of the DCB known as the ACB, or access method control block.

The OPEN routines can acquire the information they need from any of these sources, giving the user a great deal of flexibility in specifying I/O operations. To achieve device independence, for example, a user can specify a minimal amount of DCB information in the program and supply the rest of the information on the JCL for a particular execution of the program.

The OPEN routines build a data extent block (DEB), which, for DASD, specifies the device on which the volume is mounted and the physical extent of the data set on that volume. If the user program needs access method appendages or user exits to perform such functions as analyzing data errors or processing end-of-data conditions, the address of the user program and the required routines are also built into the DEB. Figure 7-7 summarizes the relationships the OPEN routines establish between the control blocks and between the user program and the access method.

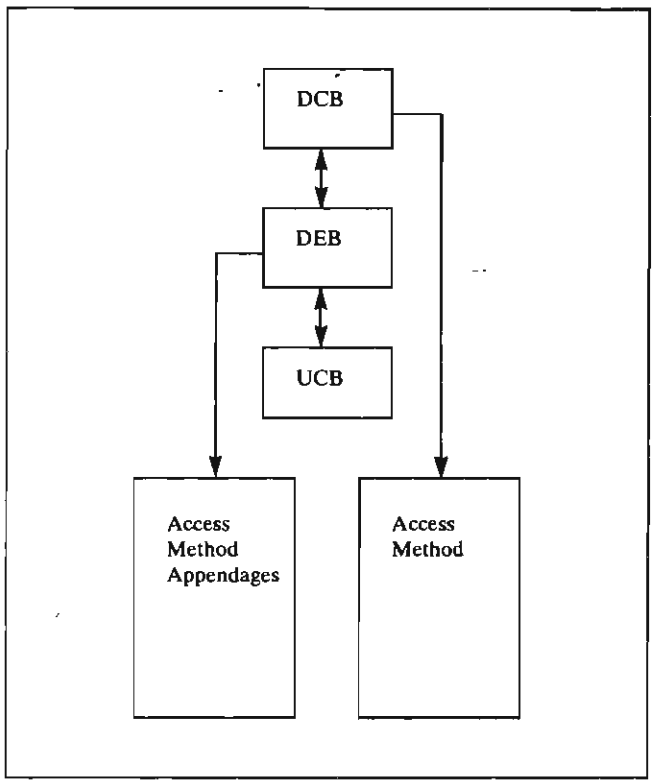


Figure 7-7. Relationships Established by OPEN

Once the data set to be used for the operation is successfully opened, it is ready to be used. The program can then issue an I/O request.

Requesting I/O

To transfer data between a data area in storage and an I/O device using an access method, the user program issues a macro instruction. GET and PUT are used for queued input and output requests; the access method does not return control to the user program until the I/O operation is complete. READ and WRITE are used for basic input and output requests; control returns to the user program once the I/O operation is initiated, and the user program must test for the completion of the operation.

Either type of request causes a branch to the access method. If the access method cannot satisfy the request because of a specification error in the request, the access method immediately returns control to the user with indicators set to describe the nature of the error. If the request was made correctly, processing of the I/O operation continues as described later in this chapter under "Access Method Functions."

Access Method Exit Appendages

Appendages are routines that enable an access method (or a user program functioning as an access method) to get control at various points during the execution of an I/O operation. Some are entered before execution of the I/O operation, others after execution, and one, the program controlled interrupt (PCI) appendage, enables a an access method to get control during an I/O operation in order to modify the channel program while it is executing.

To establish these exits, authorized routines from authorized libraries identified during system generation can be loaded during OPEN processing for authorized users. The DEB contains the pointers to the appendage routines.

CLOSE Processing

When all of the user program's requests for I/O are complete, the program must invoke the system CLOSE routines by issuing the CLOSE macro. These routines complete the final steps of the I/O operation, such as writing out the contents of the file buffer and marking the end of the file data. They also modify the DCB to break the logical connections between control blocks and between the user program and the access method. The CLOSE routines free any storage acquired by the OPEN routines.

Once the data set is closed, the user program can free the data set and the I/O device from its control with explicit JCL or program instructions. Or,the user can rely upon the the system to automatically free them at the end of the job step.

For DASD, the CLOSE routines also rewrite the DSCB for the data set to the volume. Because the DSCB can be modified during OPEN processing, a user program can change the specifications for the data set by opening and closing it.

Figure 7-8 summarizes the control blocks used as input to the CLOSE routines, the functions the CLOSE routines perform, and the control blocks that are modified during CLOSE processing.

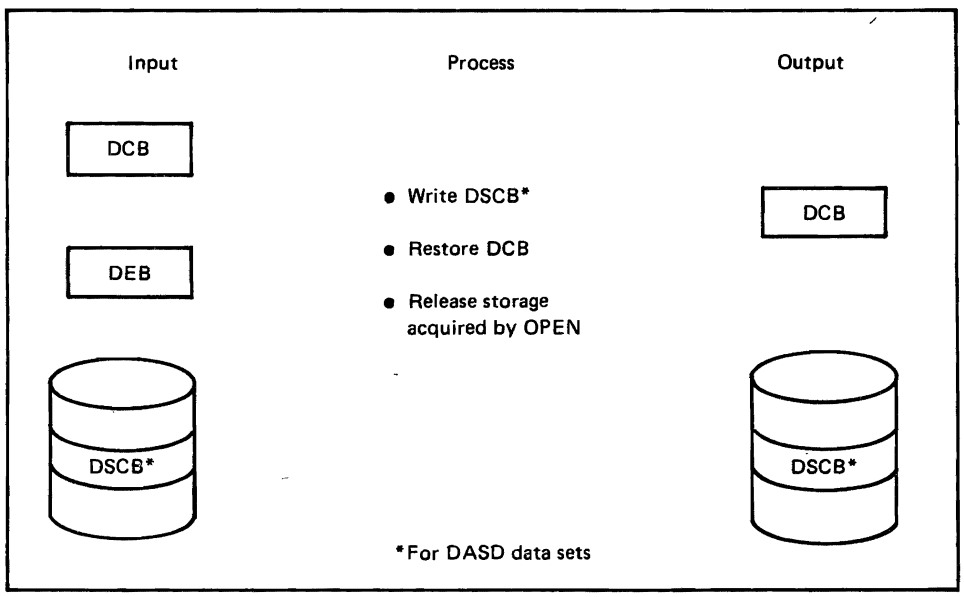


Figure 7-8. CLOSE Processing Summary

Access Method Functions

Because the OPEN routines place the address of the required access method in the DCB for the data set, the access method gets control when the user program issues an I/O macro instruction. The access method uses the control block structure built by the OPEN routines to build control blocks for the EXCP processor and a channel program for the I/O request. The access method then issues an EXCP macro instruction to pass control to the EXCP processor.

Building the Channel Program

The access method builds a channel program for the I/O operation. A channel program consists of a string of **channel command words** (CCWs) that describe the operation to the channel subsystem. Channel command words provide the channel subsystem with all of the information that it needs to perform the operation, such as the address of the data area and the number of bytes of data to be transferred.

Building Control Blocks

The access method builds two control blocks: the input/output block (IOB) and the event control block (ECB). The IOB points to the DCB; through the DCB, the EXCP processor can access the contents of the DEB and the UCB. The IOB also points to the ECB and to the channel program. The IOB thus contains pointers to all of the information EXCP and IOS need about the I/O request.

EXCP posts the ECB when the I/O operation is complete. The access method or the user program can thus test the contents of the ECB to find the outcome of the I/O operation.

Invoking EXCP

When the IOB and ECB have been built and initialized and the channel program has been created, the access method issues an EXCP macro instruction. The EXCP macro instruction causes an SVC interruption to occur. As a result of this interruption, the SVC interruption handler causes control to be passed to the EXCP processor.

Figure 7-9 summarizes the control block structure, the channel program built, and the pointers the access method establishes before it passes control to the EXCP processor.

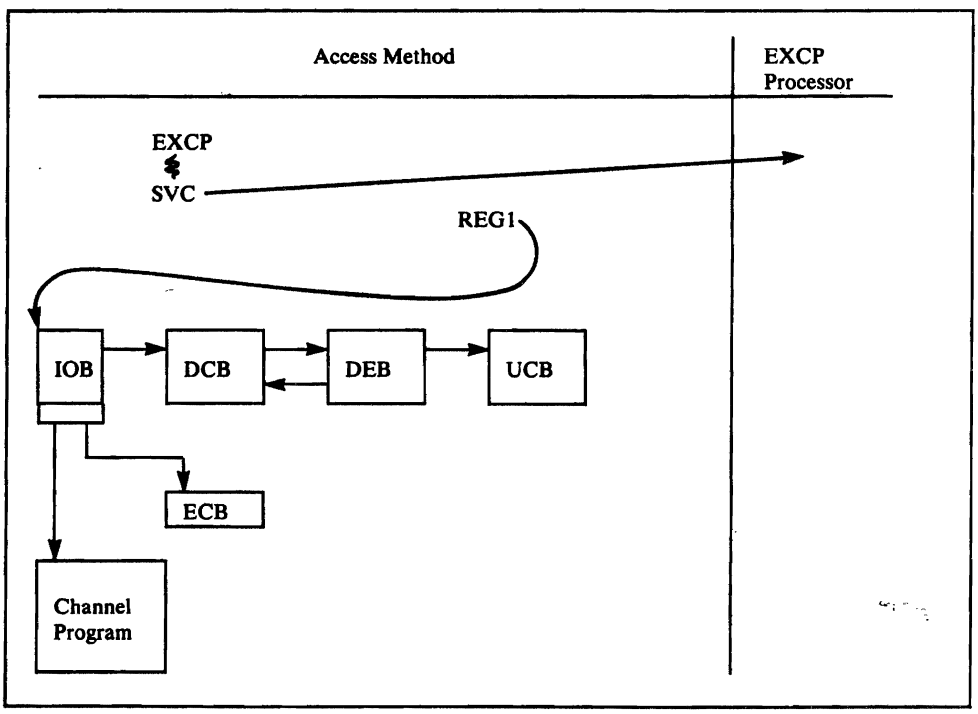


Figure 7-9. Control Block Structure for the EXCP Processor

Control returns to the access method when EXCP has sent the request to IOS. If the request used a GET or PUT macro instruction (queued access technique), the access method issues a WAIT macro against the ECB for the operation. In this case, the access method waits until the ECB is posted complete, and then it returns control to the user program. If the request used a READ or WRITE macro instruction (basic access technique), the access method returns control to the user program, which issues the WAIT macro instruction against the ECB and waits until the request is completed.

EXCP and IOS Functions

IOS is the interface between I/O requests from system components and the channel subsystem. EXCP, and other I/O drivers, pass control to IOS, and IOS, in turn, passes control to the subsystem by issuing the Start Subchannel (SSCH) instruction. IOS monitors the progress of each I/O request and the status of the I/O devices. It notifies its drivers of successful completion of a request. And, if errors occur in the channel subsystem, IOS initiates appropriate recovery actions. Because the standard access methods use the execute channel program (EXCP) processor as an interface to IOS, this chapter describes the relationship between IOS and the EXCP processor.

Figure 7-10 shows some other IOS drivers that meet the special needs of various IOS users.

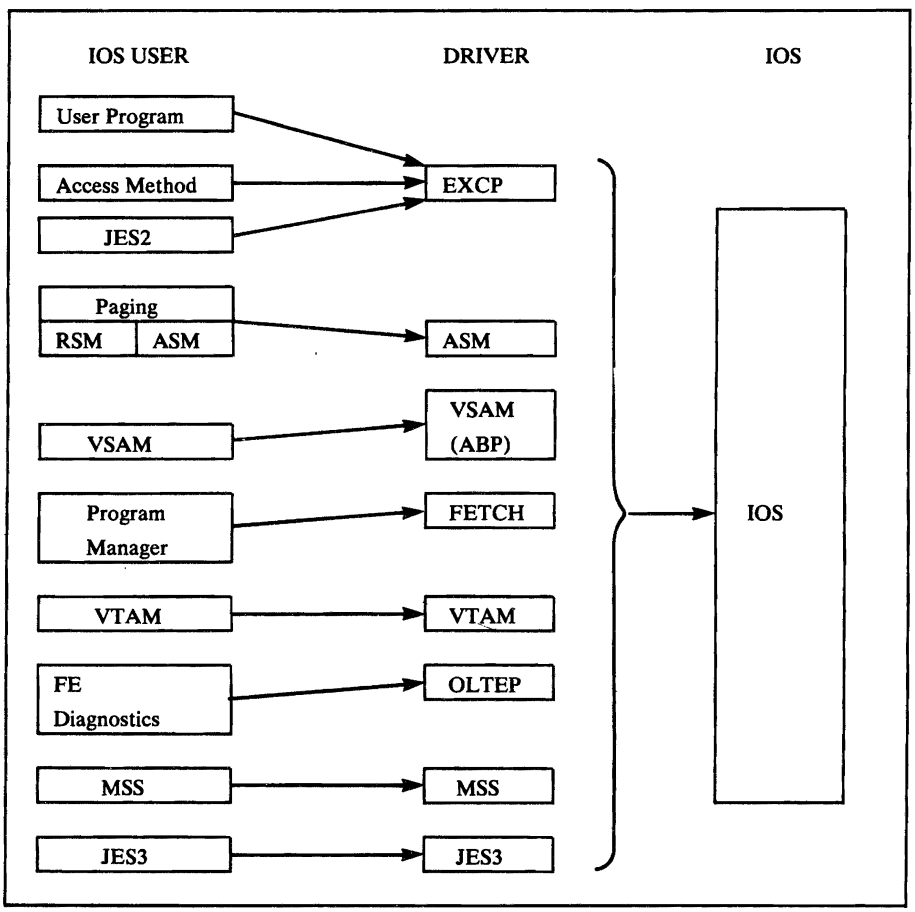


Figure 7-10. Some IOS Drivers

The EXCP processor has three major parts: front end, exit processing, and back end. These parts function in response to the needs of the I/O request to interact with the three major parts of IOS: I/O initiation, I/O interruption handling, and post status. The EXCP processor, like other IOS drivers, is separate from IOS, acting primarily as an interface between the access method and IOS. However, because the drivers and IOS work together to process a request, their functions are presented in chronological order to show the steps involved in satisfying a single I/O request.

EXCP Processor Front End

EXCP and IOS communicate by means of the I/O supervisor block (IOSB). Created by EXCP, the IOSB contains information needed to start an I/O operation such as:

- The address of the UCB for the device required by the I/O operation
- The address of the channel program translated by EXCP.

Most user programs and the standard access methods run with virtual addresses. Thus, user data areas, control blocks, and the channel programs built by the standard access methods are in virtual storage, use virtual addresses, and are pageable. However, the channel subsystem transfers data into and out of real storage locations. Therefore, the data areas, the control blocks, and the channel program for the I/O operation must be fixed and use real addresses. User programs running in virtual storage use the EXCP macro to invoke EXCP. EXCP translates the channel program and data areas to real addresses and performs **page fixing** (marks the pages as not available for page-out). Users that invoke EXCP with the EXCPVR macro provide a channel program with real addresses but use the EXCP page fixing functions.

Users that run in a real region do not require address translation or page fixing. The EXCP processor recognizes such a user and bypasses the address translation and page fixing functions.

Whether or not it performs address translation and page fixing, the EXCP front end processing constructs the control blocks IOS requires and issues a STARTIO macro instruction to activate IOS.

IOS I/O Initiation

IOS communicates with the channel subsystem by means of instructions like SSCH (Start Subchannel). The channel subsystem communicates with IOS by means of I/O interruptions. Before passing a request to the channel subsystem, IOS disables the current central processor for I/O and external interruptions and builds an I/O queue (IOQ) control block for the device requested. It also isolates the device from other I/O requests by obtaining the appropriate unit control block (UCB) lock.

IOS verifies that the subchannel for the device is usable and creates the operation request block (ORB) containing information the channel subsystem needs to process the I/O request. This information includes the address of the channel program and I/O operation control information. When IOS invokes the channel subsystem by issuing the SSCH instruction, it gives the address of the ORB as an instruction operand.

Channel Subsystem Functions

The channel subsystem executes the channel program, transfers data, updates control blocks, and when the I/O operation is complete, posts an I/O interruption. The subsystem places information about the status of the device in the subchannel information block (SCHIB) and about the completed I/O request in the interrupt response block (IRB). IOS uses the information in the IRB to determine what action to take as a result of the interruption.

The channel subsystem posts both solicited and unsolicited interruptions. Solicited interruptions result from an active I/O request on a subchannel and occur when:

1. The PCI bit in one of the channel command words of the channel program indicates a branch to a user appendage.

2. The I/O operation completes (successfully or unsuccessfully).

Unsolicited interruptions are not related to an active I/O request and occur when:

1. A device changes from the not-ready to ready state.
2. A terminal user presses the attention key.

IOS Interruption Handling

If the interruption is solicited, IOS returns control to EXCP exit processing. If the interruption is unsolicited, IOS makes tests to determine how best to handle it.

If necessary, IOS can force a device offline by **boxing** which is returning I/O requests for the device to the driver as permanent errors.

IOS operates to maintain system availability by monitoring the subchannels for:

- Hot I/O

  A Hot I/O condition is a hardware malfunction that causes repeated unsolicited interruptions from a device. IOS will either try to clear the subchannel (with the Clear Subchannel, (CSCH) instruction), take the device offline, or initiate channel path recovery routines.

- Missing Interruption Handler

  At initialization, **missing interruption handler (MIH)** control statements in SYS1.PARMLIB assign a time interval at which IOS checks the device for interruptions. If an interruption has not occurred when expected, IOS tries to resolve the missing interruption and make the device usable again. It may clear the subchannel, terminate the operation, or try the operation again.

EXCP Exit Processing

During interruption processing, IOS also recognizes and gives control to driver exits specified in the IOSB. EXCP will, in turn, give control to access method appendages provided in the DEB.

Once an interruption has been evaluated, IOS issues the SCHEDULE macro to schedule the service request block (SRB) under which the IOS post status routines run. The post status routines handle the final processing of the I/O operation. After scheduling the post status routines, IOS issues a test pending interrupt

instruction (TPI) to see if other I/O interruptions have occurred while the current one was being processed. If so, processing of the new interruption begins immediately. This action saves the time required to enable the processor for I/O interruptions, and then immediately disable it again.

MVS/XA monitors the number of TPIs issued and, if it becomes excessive, the system resource manager (SRM), might enable another processor for I/O interruptions. This facility is known as **selective processor enablement**.

IOS Post Status

The IOS post status routines complete the processing of an I/O request after the central processor has been enabled for interruptions and after EXCP exit processing completes. IOS determines what processing should be done by examining information in the IOSB about the completion of the I/O request. The post status processing can include:

- Invoking an EXCP processor exit and then returning control to the EXCP back end
- Invoking error recovery procedures (ERPs)
- Returning control to the EXCP back end

EXCP Processor Back End

The back end of EXCP issues a POST macro to post the status of the completed operation in the ECB and returns control to the dispatcher. The access method or user program that is waiting for the ECB to be posted then becomes ready for execution and is eventually dispatched. Control returns to the user program or access method at the instruction immediately following the WAIT for the completion of the I/O request.

Summary

Figure 7-11 presents an overview of the interaction between the user program, the access method, the EXCP processor, IOS, and the channel subsystem. It shows the function each performs in processing the I/O request, the instructions that pass control from step to step, and the control blocks that permit the communication of information about the I/O request.

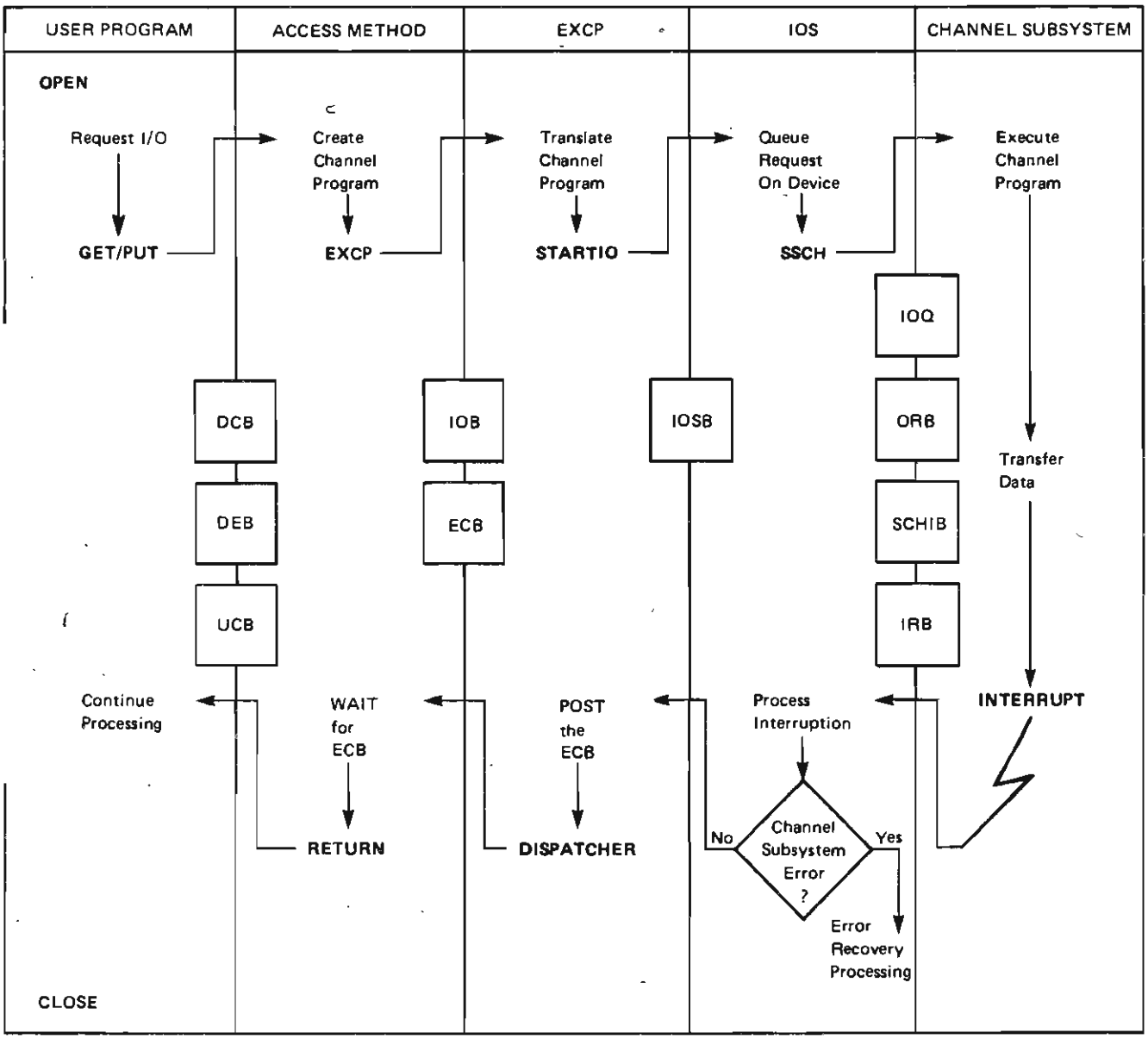


Figure 7-11. MVS/XA I/O Processing

Virtual Input/Output (VIO)

As a means of improving system performance by eliminating much of the overhead and time required to allocate a device and move data physically between main storage and an I/O device, MVS/XA provides virtual input/output (VIO). As described earlier, a physical input/output operation reads data from or writes data to a data set on an I/O device. In contrast, a virtual input/output (VIO) operation uses the system paging routines to transfer data. VIO can only be used for temporary data sets that store data for the duration of the current job; it uses the system paging routines to transfer data into and out of a page data set.

To use VIO, an installation specifies one or more I/O unit names for VIO at system generation time. Then, a user program or access method can build a channel program to send data to a system-named temporary data set on a unit that

was specified for VIO. The EXCP processor intercepts such a channel program and branches to VIO instead of invoking IOS to transfer the data over a channel to a device. VIO uses the move instruction to move that data from the channel program buffers to a special buffer in the user's address space. This special buffer is called a **window**.

The window contains enough contiguous virtual storage pages to hold all of the data that could be placed on a track for a real device. For example, a 3330 or 2305 track requires a four-page window. Figure 7-12 shows the channel program buffer and the VIO window.

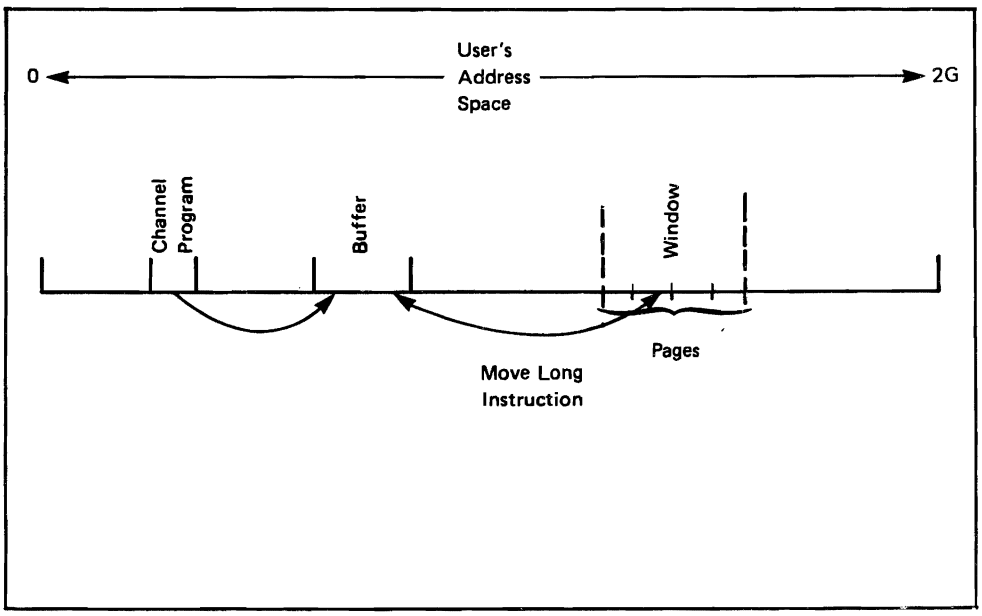


Figure 7-12. VIO Window

When the user program or access method determines that the track is full, it builds another channel program to place data on a second track. When VIO detects this track switch, it writes the contents of the window to a page data set, using the system paging routines. The system keeps VIO data set pages in real storage after this page-out, whenever possible. VIO then disconnects the window from the frames that contain the VIO data set pages. When VIO moves new data (the second track) to the window, a page fault occurs, causing fresh frames to be assigned to the window.

As the data set is created and auxiliary storage assigned, the system keeps track of the locations of each page of the VIO data set. The paging data set slots, like the real storage frames, are not necessarily contiguous; they are allocated dynamically throughout auxiliary storage as the data set is used.

When data is to be retrieved from the VIO data set, VIO locates the pages that contain the required data. If the data is not currently in the window, VIO changes the appropriate external page table entries to point to the required pages in auxiliary storage and turns on the invalid bit in the page table entries for the pages in the window. Then VIO uses the MVCL instruction to move data from the window to the channel program buffers. This instruction causes a page fault, and the proper page is either reclaimed or brought into real storage and made addressable through the window.

Thus, VIO uses paging rather than explicit I/O to transfer data. VIO eliminates the channel program translation and page fixing done by the EXCP driver as well as some device allocation and data management overhead. It also provides dynamic allocation of DASD space as it is needed. Another advantage of VIO is that the data set can remain in real storage after it is created because RSM attempts to keep the pages in real storage as long as possible.

Virtual Fetch

Virtual fetch, like VIO, is a means of improving system performance. Its goals are to streamline the process of loading modules and reduce the contention for channel paths to I/O devices.

At system initialization time, a virtual fetch address space can be created containing a directory that points to ready-to-load modules which, when they are needed, must be retrieved as quickly as possible. These modules are stored in special virtual fetch data sets on a DASD volume in an "optimized" format that minimizes the time needed to load and pass control to them. In a manner similar to VIO, virtual fetch creates a virtual storage window in the user's address space equivalent to the size of the load module so that a complete module can be loaded in one operation. Jobs in users' address spaces use cross memory services to read the virtual fetch directory in the virtual fetch address space.

Virtual fetch is useful for modules that are part of interactive applications, such as the Information Management System (IMS), where users query data files. Storing these modules in a virtual fetch data set not only reduces the time for loading them but also minimizes the competition for channel paths between the code and its data. The time needed to respond to a user request is minimized.

Access Methods

An access method is a data management routine that moves data between storage and an I/O device in response to requests made by a program. With an access method, the program is insulated from I/O details and need concern itself only with using the proper access method to meet its needs. Although the access method handles the actual I/O operation, the program using the access method still needs to be concerned with the organization of the data and the **access technique** the access method uses to move the data.

Access Techniques

There are two techniques a program can use to access the records in a data set or the contents of a message: the queued access technique or the basic access technique. Some data sets can be accessed by either technique. The access methods that support basic access and queued access techniques are logically connected to the data when the program issues the OPEN macro instruction.

The **queued access** technique is used when the sequence in which records are to be be processed is known to the access method. The system can anticipate which records are needed and make them available. When an output buffer is full, the access method writes them to auxiliary storage; when an input buffer is empty, the access method refills it.

With queued access, the program uses the GET and PUT macro instructions to transfer data. The access method automatically groups records or messages in anticipation of future I/O requests. Records or messages are then generally available when needed. Also, the access method does not return control to the

program that uses the GET and PUT macro instructions until the requested I/O operation has completed.

The **basic access** technique is used when no assumptions can be made beforehand about the sequence in which records are to be processed. The access method does not read or write a record to an I/O buffer until the program makes the I/O request.

With basic access, the program uses the READ and WRITE macro instructions to transfer data. The basic technique allows access to any records in the data set or messages from a telecommunications device. No grouping of records or messages takes place. No anticipation of future I/O requests occurs. Also, the program that uses the READ and WRITE macro instructions must test for the completion of the I/O operation because the access method returns control to the program before the I/O operation is completed.

Access Method Categories

The access methods can be viewed as falling into three categories: the conventional access methods, the telecommunication access methods, and the virtual storage access method.

Conventional Access Methods

Conventional access methods move data that resides in a data set. A data set is a collection of related records that are associated with a particular device or group of devices. If the device is a tape or a disk, the data set occupies a specific area on a volume mounted on the device drive. An MVS/XA data set can be organized in one of four ways:

- **Sequential.** Records are stored and retrieved according to their physical order within the data set.
- **Indexed sequential.** Records are physically ordered according to a key. An index or set of indexes maintained by the access method gives access to the records. Indexed sequential data sets must reside on a direct access device.
- **Direct.** The records in the data set, which must be on a direct access volume, can be organized in any way that meets the user's needs. Records are stored and retrieved according to the address of each record within the data set.
- **Partitioned.** The data set, which must be on a direct access volume, consists of members. A member is an independent group of sequentially-organized records that is accessed through its name in the directory of the data set. Partitioned data sets are generally used to store libraries of similar things, such as programs, macros, or procedures.

Access methods are usually identified by the technique they employ and the type of data organization to which they apply. For example, QSAM, the queued sequential access method, uses the queued access technique to retrieve sequentially organized records. MVS/XA supports the following conventional access methods:

- **Basic sequential access method (BSAM).** Records in a data set processed by BSAM are sequentially organized and are stored and retrieved in physical blocks. The READ and WRITE macro instructions initiate I/O operations.

The user's program must test for completion of the operation and perform any required blocking or deblocking.

- **Queued sequential access method (QSAM).** Records in a data set processed by QSAM are stored and retrieved as logical records; QSAM handles any physical blocking or deblocking required. On input, QSAM anticipates the need for a record based on its physical order; normally, the desired record is in storage, ready for use, before the request for it is made. On output, QSAM holds the logical records in a buffer and performs physical output only when the buffer is filled.

- **Basic direct access method (BDAM).** Records in a data set processed by BDAM can be organized in any manner chosen by the programmer. The data set must reside on one or more direct access volumes. Records are stored and retrieved by actual or relative addresses within the data set.

- **Indexed sequential access method (ISAM).** Records in a data set processed by ISAM are arranged in sequential order according to the contents of a key. ISAM maintains an index structure that is used to locate a particular record. Access to the records can be either sequential (QISAM) or direct (BISAM). Both the data set and the indexes must reside on a DASD volume.

- **Basic partitioned access method (BPAM).** A data set processed by BPAM consists of a number of members and a directory that holds the name and location of each member. A member contains a group of records that are organized sequentially. BPAM maintains and accesses the directory; once BPAM locates the desired member, the records within the member are processed by BSAM or QSAM. The data set, including the directory, must reside on a DASD volume.

Telecommunication Access Methods

Telecommunication access methods move data as messages. A message is a collection of related pieces of data sent and received as a single unit between the remote device and storage. If the remote device is an interactive terminal, the data in the message is the data the terminal user enters at the keyboard and sends to the application, or the data that the application sends to the terminal for display or printing. The terminal or access method turns this data into a message by embedding in it standard communications line control information, and the modems further convert the message characters into a form suitable for transmission over the data link.

MVS/XA provides three access methods for moving data over telecommunication I/O paths between storage and the I/O device:

- **Basic telecommunication access method (BTAM).** The READ and WRITE macro instructions move messages between storage and the device. BTAM manages the messages it processes across all the various communication lines being used.

- **Telecommunication access method (TCAM).** The GET or READ macro instructions and the PUT or WRITE macro instructions move messages between storage and the device. TCAM allows an application to perform its own message routing, message editing, and error checking.

- **Virtual telecommunication access method (VTAM).** Data transfer between the application and the terminal occurs in either record mode or basic mode. In **record mode**, the application issues SEND and RECEIVE macro instructions to transmit data between the terminal and storage. In **basic mode**, the application issues READ and WRITE macro instructions to transmit messages between the terminal and storage.

VTAM is the primary access method used to support the system network architecture (SNA), an overall system definition of the functional responsibilities of telecommunication system components upon which new teleprocessing applications can be planned and implemented.

Virtual Storage Access Method (VSAM)

The virtual storage access method (VSAM) is specifically designed to take advantage of virtual storage. VSAM is for access to DASD data and runs in virtual storage and uses virtual storage to buffer I/O operations. VSAM is one access method that does not use the EXCP processor.

VSAM employs modified queued and basic access techniques and can process three types of data sets: key-sequenced, entry-sequenced, and relative record. The order in which the data set is initially loaded and updated is different for each type.

For a **key-sequenced data set**, records are loaded, as the name implies, in key sequence. Each record must have a key, and the ordering of the records is determined by the collating sequence of the keys. Any new records subsequently added to the data set are added in key sequence.

For an **entry-sequenced data set**, records are loaded in sequential order as they are entered. New records are added at the end of the data set.

For a **relative record data set**, records are loaded according to a relative record number that can be assigned either by VSAM or by the user program. When VSAM assigns the relative record number, new records are added at the end of the data set. When the user program assigns the relative record number, new records can be added in relative record number sequence.

Chapter 8. Entering and Scheduling Work

MVS/XA processes an installation's workload as jobs. A **job** can be viewed as a series of job control language (JCL) statements. JCL identifies the program to run, and information such as what data sets, devices, or other system resources the job needs when it runs. The input data needed by a job may be included with its JCL, or the JCL may refer to data in existing data sets.

A collection of jobs presented to MVS/XA in this way is called an **input stream**. Each user classifies a job in an input stream by assigning it a job class. A **job class** is defined by the installation. Jobs of similar characteristics and processing requirements are generally assigned to the same job class. For example, long-running data processing programs may require setting up many DASD or tape volumes and disrupt the turnaround time for a daily workload such as invoice and accounts receivable processing. The long-running jobs can be assigned to a single job class, and MVS/XA can process them when the system is not busy and when the resources they need are available.

A user also classifies each job's output by output class. An **output class**, which is defined by the installation, is used to describe the output on local or remote printers or punches or to schedule output through the subsystem interface. Grouping output with similar characteristics by output class allows MVS/XA to keep the existing system output devices as active as possible.

Other installation-specified job characteristics also help MVS/XA use system resources effectively. A job's priority is an important one. If MVS/XA knows the priority of each job, it can order its processing of jobs, running high priority jobs before low priority jobs.

The role of the Job Entry Subsystem

For reasons such as the efficient use of system resources, MVS/XA breaks a job into tasks and processes each task separately. At any point in time, the computer system resources are busy processing the tasks for various jobs. Other tasks are queued awaiting resources. Actually, MVS/XA divides the management of jobs and resources between the job entry subsystem and MVS/XA components. Generally speaking, the job entry subsystem manages jobs before and after execution; MVS/XA manages them during execution.

Thus, an MVS/XA installation requires a job entry subsystem (JES) in order to process jobs. Its function is to screen jobs before admitting them to the system and to handle their termination when processing is done. JES ensures that the job request has been properly made and translates it to the correct form and places in the right category for processing under MVS/XA.

The job entry subsystem reads an input stream. It reads each job and places it on a direct access device (or devices) known as the **spool** device. SPOOLing, or simultaneous peripheral operations on-line, is the temporary storing of jobs and job-related data in intermediate stages of processing so that they are readily accessible. Because each job has a job class, priority, and output class, the job entry subsystem selects jobs from the spool device for execution in a way that encourages the effective use of other system resources.

This chapter describes in more detail what the job entry subsystem does, how it ensures that system resources are allocated to the job, and how it works in various

MVS/XA environments. The way MVS/XA specifically controls a job once it is selected for execution is described in Chapter 6, "Supervising the Execution of Work."

Job Entry/Output Processing

Even though each MVS/XA system uses only one job entry subsystem, there are actually two IBM job entry subsystems available: JES2 and JES3. They differ in three important respects: how they select a job to be scheduled for execution, how they allocate resources for a job, and how they control the spool in multiprocessing systems. These differences are described in more detail as this chapter proceeds.

Job entry subsystem processing includes the entry and output of jobs and occurs in six stages:

- Entry
- Conversion/Interpretation
- Device allocation
- Scheduling a job for execution
- Output
- Purge

The following descriptions of the stages of job entry and job output processing generally apply to either MVS/XA job entry subsystem. When necessary, they indicate any processing uniquely performed by either JES2 or JES3.

Entry

The job entry subsystem reads an input stream from a device such as a card reader, remote terminal, another MVS or MVS/XA system, tape drive, or direct access device.

Users at remote work stations as many as hundreds of miles from the job entry subsystem can submit jobs by means of **remote job entry (RJE)**. A work station may be a single I/O device, a number of separate devices, or one of a number of allowable non-system/370 processors with their devices. The job entry subsystem can write the output of a remotely submitted job on local devices or transmit it to any work station connected to the job entry subsystem.

There are two methods for RJE communication: **Binary synchronous communication (BSC)**, where each device at a work station needs a separate communication line; and **system network architecture (SNA)**, where many devices can share a line. JES2 and JES3 use both BSC and SNA for remote job processing.

Jobs, themselves, can create input streams. Rather than being entered from a device, job-created input streams are processed by a JES internal reader program. An **internal reader** includes a special data set that other programs can use to submit jobs, control statements, and commands to the job entry subsystem. Any job executing in MVS/XA can use an internal reader to pass an input stream to the job entry subsystem, and the job entry subsystem can receive multiple jobs concurrently through multiple internal readers.

During system initialization, for example, MVS/XA uses two internal readers, to pass the JCL for started tasks, MOUNT commands, and TSO LOGON requests to the job entry subsystem. They are:

- STCINRDR, which the started task control (STC) routine uses to process a START or MOUNT command. When starting VTAM, for example, STC creates the JCL to run the VTAM procedure and passes this JCL to the job entry subsystem through the STCINRDR internal reader.

- TSOINRDR, which is used by the TSO LOGON command to initiate a TSO terminal session. The LOGON command generates a job identifying the user's logon procedure. The job entry subsystem reads this job from the TSOINRDR internal reader.

As the job entry subsystem reads the input stream, it assigns a job ID to each job and places each job's JCL, optional JES control statements, and input data into spool data sets. Jobs are then selected from the spool for processing and subsequent execution.

Batch jobs are selected by the job entry subsystem in response to request for work from the initiator function of the MVS/XA job scheduler. They run in the initiator's address space. Jobs created by TSO LOGON, the MOUNT command, or the START command are selected for processing when they are entered by a process known as **demand select**. These jobs run in their own address spaces. No matter how they are selected, and regardless of the address space in which they run, once they have been selected, all jobs are processed in much the same way.

Conversion/Interpretation

The job entry subsystem uses a converter program to analyze each job's JCL statements. The converter takes the job's JCL, merges it with JCL from a procedure library (usually SYS1.PROCLIB), and converts the composite JCL into internal text (a form of data that the job entry subsystem and the job scheduler functions of MVS/XA both recognize). If the converter detects any syntactic errors in the JCL, it issues diagnostic messages and places the job on the output queue; the job won't be selected to run.

If the job has no syntactic errors, JES2 stores the internal text in a spool data set and queues the job for execution according to its priority within its job class. When JES2 finally selects a job for execution by MVS/XA, the interpreter function will further analyze the JCL and build control blocks. JES3, in contrast, invokes the interpreter at the outset and stores both the internal text and control blocks in the spool data set. JES3 may also perform additional processing before scheduling the job for execution. The sections that follow explain when this occurs.

Device Allocation

Most jobs have auxiliary storage requirements. That is, a job generally needs to use I/O devices, such as tapes or DASDs, and data sets when it runs. MVS/XA assigns these resources to jobs through a function called device allocation. Device allocation uses the information in the job's JCL statements to assign the proper resources — devices, volumes, and data sets — to the job.

Each job's JCL statements identify the job (JOB statement), each job step within the job (EXEC statement), and the data sets to be used by the job (DD statements). A job can have one step (single EXEC statement) or multiple steps

(multiple EXEC statements). Each EXEC statement is normally followed by DD statements that identify the data sets that are to be allocated for use by the job step. The parameters on the DD statement identify such things as:

- The name of the data set
- The name of the volume on which the data set resides
- The type of I/O device that holds the data set
- The format of the records on the data set
- Whether the data set exists or is to be created
- The size of the data set to be created

Device allocation uses this information to identify the devices, volumes, and data sets to be used by the job steps and to assign them to the job step so that ❶ those devices, volumes, and data sets that can be shared are available to other job steps and ❷ those devices, volumes, and data sets that cannot be shared are used only by this job step. Through device allocation, MVS/XA tries to ensure that no job step that is ready to execute has to wait for its devices, volumes, or data sets to be assigned.

Device allocation performs the following general functions to allocate resources:

- Locating the volume and unit information for a requested data set
- Resolving relationships among two or more requests
- Creating, through data management, new data sets
- Assigning I/O devices to the request
- Instructing the operator to mount necessary volumes
- Allowing dynamic concatenation and deconcatenation of data sets

Device allocation performs the following general functions to deallocate resources:

- Controlling what happens to a data set when a job step finishes using it
- Releasing a data set, reserved by an initiator, for use by other job steps
- Releasing I/O devices for use by other job steps

MVS/XA has three forms of device allocation to assign resources to jobs:

- **Job step allocation:** The initiator allocates devices as part of initiating a job step. (An **initiator** is an MVS/XA system program that the operator starts or that JES2 or JES3 starts when the system is initialized. Its function is to start execution of a job step. An initiator starts a job step by allowing it to compete for system resources with other jobs that are already running.) Job step allocation is used by JES2 and JES3 and when the TSO LOGON, START, or MOUNT command enters a job.

- **JES3 device allocation:** JES3 allocates devices before passing a job to the initiator.

- **Dynamic allocation:** A job allocates devices as it executes.

Job Step Allocation

Job step allocation consists of various system allocation routines that analyze the DD statement information for each job step. JES2 is the primary user of job step allocation; JES3, as will be described later, can perform many allocation functions itself before the job begins and the MVS/XA allocation routines execute.

After JES2 selects a job to run and passes it to the initiator, the initiator invokes the interpreter to create scheduler work area (SWA) control blocks that describe the job's resource requirements. The initiator then passes control to the system allocation routines for the first step in the job. The system allocation routines use the SWA control block information to analyze the job's device, volume, and data set requirements and allocate those resources needed by the program for that job step. The initiator does not start the job step until the system allocation routines assign all the resources the job step needs. When all resources are ready, the system allocation routines return to the initiator, which starts the job step. After the job step finishes running, the initiator uses the system deallocation routines to release those resources no longer needed; the initiator then repeats its use of the system allocation routines for the next job step.

JES3 Device Allocation

A user whose job is processed by JES3 can use JES3 device allocation to allocate resources before the job is selected to run. The user controls the extent to which JES3 allocates devices, volumes, and data sets to the job. At one extreme, the user can bypass JES3 device allocation altogether. At the other extreme, the user can have JES3 allocate devices, volumes, and data sets for all of the steps in the job before the job is selected to run. In either case, JES3 reads the SWA control blocks for the job from the spool data set and passes them to the initiator when the job is selected to run. The initiator invokes the system allocation routines of job step allocation. These routines analyze the SWA control blocks and endorse the allocation decisions already made by JES3, or they assign required devices, volumes, or data sets that have not yet been allocated to the job.

Three categories of devices can be defined for the JES3 installation:

- JES3 devices, which are exclusively managed by JES3
- JES3 and MVS/XA devices, which are jointly managed by JES3 and MVS/XA
- MVS/XA devices, which are exclusively managed by MVS/XA

JES3 can take an active role in assigning the devices it exclusively manages and the devices it jointly manages by:

- Selecting certain jobs over other jobs competing for resources in order to keep each processor as busy as possible. For example, JES3 normally selects for execution on a given processor the first job (within a given priority) that can acquire the resources it needs on that processor.

- Selecting an eligible processor in a multi-processing complex on which to allocate devices for a selected job. JES3 compares each job's resource requirements with the JES3-managed devices attached to each processor. JES3 selects the processor with the best match of shareable devices. This emphasis on shareable devices helps to increase the number of concurrent device allocations that can be performed, thus increasing the number of jobs that can be processed concurrently.

- Assigning devices, volumes, and data sets to jobs to maximize the use of the devices and minimize the physical movement of volumes.

A JES3 installation can also define a pool of devices (called device fencing) to be used exclusively by a set of jobs in a job class group. In addition, the installation can optionally allow this set of jobs to use devices not in this pool and have other

devices allocated as needed. **Device fencing** lets the installation tailor its device use to its workload.

Dynamic Allocation

Because resource requirements might not be fully known before execution, dynamic allocation routines are available to enable jobs and time-sharing users to acquire resources as the need develops. Dynamic allocation also allows resources to be used more efficiently because the resources can be acquired just before use and released immediately after use.

A typical use for dynamic allocation occurs in a program that needs temporary use of a device, volume, or data set for which there is heavy contention. In such a case, dynamic allocation provides the means for a job to tie up the resource for only as long as necessary rather than for the life of the job.

Another common use for dynamic allocation is in a job whose need for allocated resources might vary according to its input. Dynamic allocation permits such jobs to dynamically allocate and free only the data sets necessary to process the input, so the specific resources supporting the required data set can be in use for the minimum time. A job can use dynamic allocation to free a SYSOUT data set so that the job entry subsystem can process it while the job is still executing. Such data sets are called **spin-off data sets**.

Scheduling a Job for Execution

The execution phase of the job entry subsystem responds to requests for jobs from the MVS/XA job scheduler initiator function. The job entry subsystem selects jobs from a job queue on a spool data set and sends them to this function. The job queue contains jobs in the following stages of processing:

- Jobs waiting to run
- Jobs currently running
- Jobs waiting for their output to be produced
- Jobs having their output produced
- Jobs (for which all processing has completed) waiting to be purged from the system.

By distinguishing among jobs on the job queue, the job entry subsystem can manage the flow of jobs through the system. JES2 and JES3, however, schedule jobs in different ways.

JES2 Job Scheduling

To process the jobs on the job queue, JES2 communicates with an initiator. The initiator asks JES2 for a job. JES2 knows what job class or job classes are assigned to the initiator and in what order the job classes should be searched for a job. If the initiator, for example, is assigned two job classes, JES2 scans the job queue to determine if any jobs in the first class are waiting for execution before scanning the job queue for any jobs in the second class. Within a given class, JES2 selects jobs according to their priority. JES2 selects the lowest priority job in the first class ahead of the highest priority job in the second class. It selects jobs from the second

class only when there are no jobs in the first class. When JES2 selects a job it passes it to the initiator.

Associating each initiator with one or more job classes in this way allows an installation to control job selection to encourage a more efficient use of available system resources. Assume, for example, the following job class assignments exist:

Class B = jobs that need special devices
Class C = jobs with high instruction processing requirements
Class D = jobs with high I/O-request requirements

Assume also that the following initiator assignments apply:

Initiator 1 can process classes B, C, and D
Initiator 2 can process classes C, D, and B
Initiator 3 can process classes D, B, and C

Initiator 1 can accept jobs in classes B, C, and D, but will process class C jobs only when class B is empty, and class D jobs only when classes B and C are empty. If there are jobs on the queue in all three classes and all necessary resources (for example, I/O devices and data sets) are available, then three jobs (one from each of the three different classes) can run concurrently Each initiator runs the highest priority job in its highest priority class.

After JES2 selects the highest priority job in a job class for the initiator and passes the job to it, the initiator invokes the interpreter to build control blocks from the internal text that the converter created for the job. The interpreter builds these control blocks in the scheduler work area (SWA) of the initiator's address space.

The initiator then allocates the input and output devices specified in the JCL for the first step of the job. This allocation ensures that the devices are available before the job step starts running. The initiator then starts the program requested in the EXEC statement.

JES3 Job Scheduling

To process a job on the job queue, JES3, like JES2, communicates with an initiator. While JES2 relies on the installation to control the job mix through its assignments of job classes to initiators, JES3 job scheduling algorithms control the job mix in order to provide the correct proportion of I/O-bound and processor-bound jobs. To control the job mix, JES3 uses predefined job class groups.

JES3 associates a **job class group**, a set of job classes, with one or more initiators and also with specific devices and processors. The installation defines job class groups during JES3 initialization; this definition allows JES3 to control:

- The maximum number of jobs of a given class that can be readied to run
- The maximum number of jobs that can run in the JES3 installation
- The maximum number of jobs that can run on a given processor at one time
- The resources a job uses, such as initiators, storage, and devices

- The kind of job selection and job priority adjustments allowed for jobs waiting to be selected to run

After JES3 prepares a job to run, it passes the job to the initiator. The initiator can normally activate the job immediately because JES3 has allocated the devices this job needs. Once the job is running, the MVS/XA allocation routines perform any additional device allocations that are needed.

Additional Job Scheduling Functions

When all initiators are busy, the progress of certain jobs through the system may fall below normal expectations. To help in these situations, JES2 and JES3 perform additional scheduling functions that attempt to reduce the time required to schedule jobs, that help to ensure that certain jobs are selected to run by a certain time, and that schedule jobs dependent on the success or failure of other jobs. These scheduling functions are:

- Execution batch scheduling (JES2)
- Deadline scheduling (JES3)
- Priority aging (JES2 and JES3)
- Dependent job control (JES2 and JES3)

Execution batch scheduling is an extension of normal JES2 job scheduling that helps to increase throughput by reducing the job scheduling overhead for certain types of jobs. Jobs eligible for execution batch scheduling are jobs of relatively short duration, especially single-step jobs that have common device setup requirements and jobs that are run frequently. Examples of such jobs are compile-and-go, debugging, order-entry, and file-inquiry jobs.

To use the execution batch scheduling facility, an installation must write an execution batch (XBATCH) processing program and a procedure to initiate it, and assign the jobs a unique job class associated with the execution batch procedure. Also the installation must include execution batch scheduling parameters when initializing JES2.

When JES2 recognizes a job with the execution-batch-scheduling job class, JES2 builds and processes JCL to invoke the XBATCH procedure. Once the XBATCH procedure initiates the XBATCH program, the program remains active as long as it has jobs to process. Thus execution batch scheduling involves gathering related jobs into a single input stream and passing them as an input data set to the user-written XBATCH program. This process reduces the initiator's overhead associated with setting up for and processing numerous individual jobs or job steps.

Deadline scheduling allows a JES3 installation to specify a time of day (deadline) by which a given job should be selected to run or a job's output should be scheduled. A job requests deadline scheduling and specifies the deadline time through JES3 control statements in its JCL. If the job remains in the job queue as the deadline approaches, JES3 increases the job's selection priority — that is, the priority at which the job is selected to run — until the job is selected to run or until a maximum priority is reached. The operator can modify the parameters that affect deadline scheduling in order to deal with unforeseen changes in the installation's workload.

Priority aging ensures that jobs that have been waiting to run in the workload of either a JES2 or JES3 installation have a chance of being selected to run before

those jobs that just entered the system. JES2 and JES3, however, differ in how they implement priority aging.

JES2 can increase the priority of a job within its job class depending on the length of time the job has been in the system. By using priority aging, a JES2 installation can increase the priority of a waiting job. The longer the job waits, the higher its priority becomes and the greater its chances of being selected to run. JES3, on the other hand, increases the priority of a job depending on the number of times the job has been passed over for selection.

Dependent job control (DJC) is a JES3 function that allows jobs to run in a predefined order. That is, the user can specify that one set of jobs be completed before another job or set of jobs. Also, devices used by a set of jobs under dependent job control can be reserved for those jobs in that set, ensuring that they'll be available when needed. The Chained Job Scheduler (CJS) is an IBM program product that provides a similar function for JES2 users.

Output

The job entry subsystem controls all SYSOUT processing. While running, a job can produce system messages that must be printed, as well as data sets that must be printed or punched. After the job finishes, the job entry subsystem analyzes the characteristics of the job's output in terms of its output class and setup requirements and processes its output accordingly. Specifically, the job entry subsystem gathers the output data by output class, device availability, process mode, and set-up characteristics, then queues it in the SYSOUT data set on the spool device for output processing.

MVS/XA includes an external writer program (XWTR) that uses the subsystem interface for SYSOUT processing. An installation uses this external writer to write to devices other than those supported by the job entry subsystem. Installation written external writer programs can also control the output; these programs tailor the output to the installation's needs.

Purge

When all processing for a job is completed, the job entry subsystem releases the spool space assigned to the job, making it available for allocation to subsequent jobs. The job entry subsystem also issues a message to the operator to indicate that the job has been purged from the system.

Job Entry Subsystems in a Multi-System Environment.

For an installation with a single MVS/XA system, JES2 and JES3 perform the same basic functions. That is, they read jobs into the system, convert them to internal form, select them for execution, process their output, and purge them from the system. But, for an installation having more than one MVS/XA processor or processor complex in the configuration, there are noticeable differences between JES2 and JES3 processing. Figure 8-1 and the discussion that follows illustrate these differences:

1. Control of job entry processing

   JES2 exercises **independent control** over its job processing functions. Each JES2 processor controls its own job input, job scheduling, and job output processing.

Each JES2 multiple system configuration, also called a **multi-access spool configuration**, or node, consists of two or more JES2 processors at the same physical location, all sharing the same job queue and spool. Each JES2 processor can read jobs from local and remote card readers, select jobs from the shared spool for execution, print and punch results on local and remote output devices, and communicate with the operator.

The common job queue enables each JES2 processor to share in processing the installation's workload; jobs can execute on whatever processor is available and print or punch output on whatever processor has an available device with the proper requirements. One JES2 processor can process a job's input while another JES2 processor may schedule and execute the same job.

If one processor in the configuration fails, the others can continue processing from the shared job queue. Only work in process on the failed processor is interrupted; the other JES2 processors continue their processing.

JES3, in contrast, exercises **centralized control** over its job processing functions. JES3 controls the job input, job scheduling, and job output processing in a single processor complex, called the **global** JES3 processor. Other JES3 processors attached to the global processor are called **local** processors and are under the control of the global JES3 processor. The global JES3 processor and each local JES3 processor form a loosely-coupled multiprocessing configuration; they communicate over a channel-to-channel (CTC) adapter, which carries control information between the global and local processors. Together, the global and local processors comprise a JES3 node.

As with JES2, each JES3 processor can access the spool data set, which consists of SYSIN and SYSOUT data, JCL, and the job queue for the entire JES3 installation. It is the JES3 global processor, however, that reads jobs from local and remote input devices, places them on the spool, and selects them to run on the global or any local processor; the global processor controls all the processing of job output. The local processors access the spool only to read or write data for jobs executing on the local processor.

The JES3 system operator can dynamically bring up a local JES3 processor as the JES3 global processor if the global processor fails. The relink of this new JES3 global processor to the remaining local JES3 processors is performed automatically. Jobs that were executing on the failed processor can be recovered.

2. Selection of jobs for processing

Both JES2 and JES3 process jobs that have been read into the system and placed on the spool. Each JES2 processor has access to the spool and independently selects jobs for processing from the spool. In contrast, only the global JES3 processor selects jobs from the spool for processing even though all JES3 processors share the spool. As a result, the JES3 environment can control the integrity of shared data sets. A JES2 installation can provide a similar degree of control by combining its JES2 systems into a **global resource serialization complex**. As explained in Chapter 6, "Supervising the Execution of Work," a global resource serialization complex with the systems connected by channel-to-channel adapters can protect the integrity and increase the availability of data on DASD shared among all of the processors in the complex.

3. Multi-processor configurations

   Both JES2 and JES3 multi-system configurations may combine a variety of multiprocessors, such as a 3081 dyadic processor complex and a 3084 four-way processor complex. For JES2, there can be no more than seven operating systems in the configuration. JES3 is limited to eight operating systems although the actual capacity might be less depending upon the work mix in the configuration. A loosely-coupled configuration becomes more difficult to manage as the number and speed of the MVS/XA systems within it increase.

4. System operation

   JES3 presents the system operator with a **single system image**. There is one JES3 console to which all of the 96 routing codes direct JES3 messages. (If a JES3 console failure occurs messages can be routed to an MVS/XA operator console). The JES3 operator need not be concerned about where work comes from or where it is processed.

   JES2 presents a less unified system image. JES2 system message traffic, for example, is sent by means of 16 routing codes to various consoles according to the functions they affect.

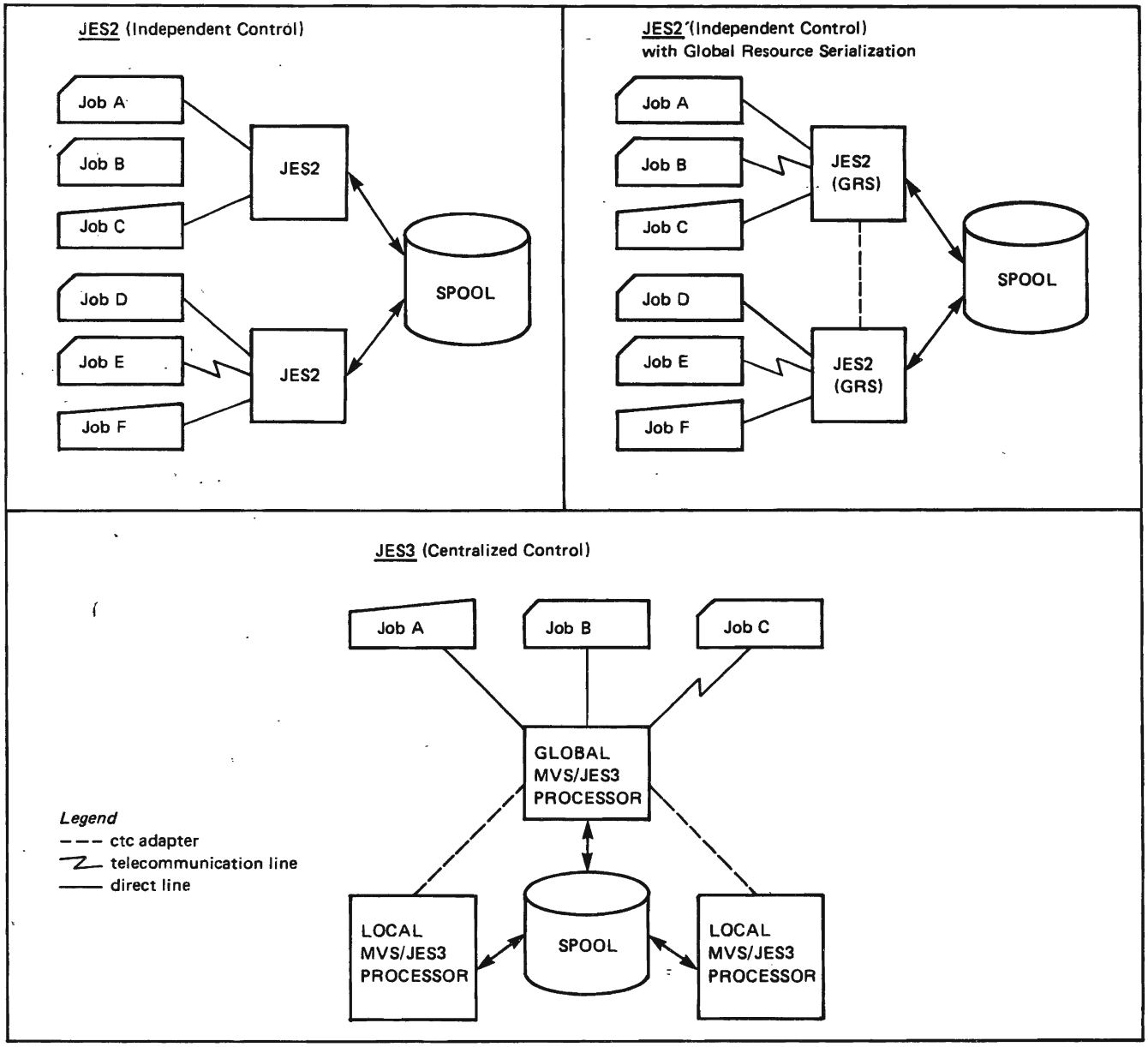


Figure 8-1. Job Entry Subsystem Configurations

Job Networking

Job entry subsystem nodes, each at different physical locations, can be joined through communication lines (such as those used for telephone or satellite communications) or channel-to-channel (CTC) adapters to form a **network**. A job entered at one location can be transmitted to another location in the network where it can use, for example, special hardware, or software features, a centralized data base, or special applications. Similarly, reports produced by an accounting program, for example, can be distributed automatically to several locations in the network.

JES2 nodes in a network use the network job entry (NJE) function that is a part of JES2 to process jobs. The NJE facility enables JES2 to:

- Manage the paths between the JES2 nodes joined in a network so that work moves from place to place, a process known as **automatic path management**.
- Transmit and receive input streams, commands, messages, and output among JES2 nodes in the network.
- Allow the system operator at any node to control jobs throughout the network.

With NJE, each JES2 node in the network can process jobs from other JES2 nodes. JES2 nodes can pass both jobs and job output among themselves for processing. The installation can choose between either SNA or BSC communications protocols for a link between two nodes within a JES2 network. For the network as a whole, the installation can use a combination of SNA and BSC links.

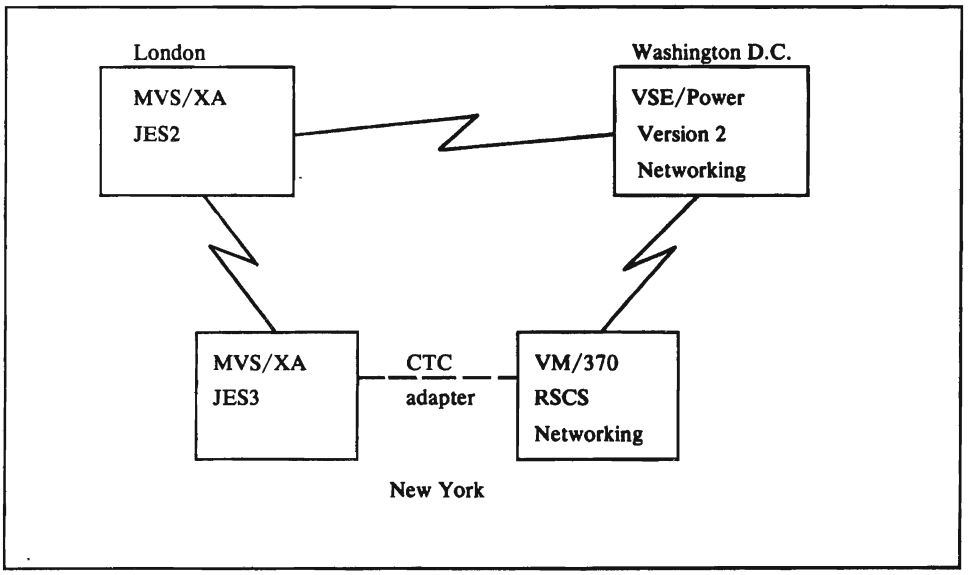


Figure 8-2. Job Networking

A JES2 NJE network can also be extended to include VSE/POWER Version 2 with VM/370 and JES3. The remote spooling communications subsystem (RSCS), for example, allows a VM/370 node to participate in job networking with a JES2 NJE node. Figure 8-2 illustrates a network.

JES3, like JES2, also contains a network job entry function to allow job networking. A network of JES3 nodes can be joined together by communication lines. Each global JES3 processor in the network communicates with other global JES3 processors at other JES3 nodes (there is no master-subordinate relationship), offering advantages similar to those that JES2 NJE offers; jobs can be submitted at one location and executed at another, and job output can be produced at any location within the network. JES3 uses binary synchronous communications (BSC). In addition to JES3 nodes, a JES3 network can also be extended to include VSE/POWER Version 2 Networking with VM/370 and JES2. These non-JES3 nodes then participate with JES3 nodes in job networking.

Comparing JES2 and JES3 Features

FEATURE	JES2	JES3
Device Allocation	NO	YES
Job Scheduling		
Execution batch scheduling	YES	NO
Deadline scheduling	NO	YES
Priority aging	YES	YES
Dependent job control	YES	YES
SPOOL Management	Independent	Dependent
Off-line backup	of JES2 Release	on JES3 Release
RJE: Both SNA and BSC	YES	YES
Single System Image		
Single job queue	YES	YES
Data integrity	YES	YES
Console	NO	YES
Device fencing	NO	YES
Networking		
BSC	YES	YES
SNA	YES	NO
Automatic path management	YES	NO
Console Routing Codes	16	96
Automatic Operator Commands	YES	NO
Performance Monitoring	NO	YES
Maximum Component Systems	7	8

Figure 8-3. JES2 and JES3 Features

Chapter 9. Monitoring System Activity

The first operating systems were simple to use and fix, yet inefficient in several ways. Long-running jobs held up other jobs, and only those resources associated with the active program were used. All other resources waited. This inefficiency derived from the system's simple operation. On the other hand, the system's simple operation had specific benefits. When there was a system error, it was generally easy to determine what program was executing at the time. Also, accounting algorithms for charging users involved simple computations (job stop time minus job start time). Using the system efficiently was more a matter of establishing efficient installation procedures for processing jobs rather than using sophisticated operating system function to handle the job-to-job transition.

In contrast, MVS/XA is not a simple system, yet its design keeps more work going on in parallel. More interruptions occur. More task switches take place. More resources are shared. More non-serial operation occurs. MVS/XA does these things through sophisticated control programs - programs that dispatch work, that save job status, that switch from one piece of work to another, that keep things straight among the many programs that share common resources, and that read jobs into the system and produce their output in parallel with controlling the jobs already in execution.

MVS/XA, like earlier operating systems, handles job-to-job transition. However, MVS/XA handles a variety of job types. A job can be part of a batch input stream, an interactive terminal session, or an installation program that runs in the background (low priority) or foreground (high priority) of the system's work. Moreover, a single job is generally not as easy to identify because MVS/XA splits each job into pieces. The job entry subsystem for example, processes a job as records on the spool, the dispatcher as address spaces, TCBs, and SRBs, the interruption handlers as status save areas, and the system resources manager as swapped-in or swapped-out address spaces.

As a result, jobs lose much of their identity. The single job, started, executed, and completed, is a collection of individual pieces of work efficiently dispatched, interrupted, redispatched, and eventually completed. An MVS/XA job, then, equals all the completed pieces of work. So when a job fails in MVS/XA, the diagnosis must focus on locating the piece that failed. And, because of MVS/XA's complexity, finding this piece can be difficult.

MVS/XA helps to make this diagnosis easier by providing various monitoring mechanisms that can keep track of the individual pieces of work in the system. These monitoring mechanisms condense the pieces of work into a processing history the installation can use to isolate, diagnose, and fix program errors.

Other MVS/XA monitoring mechanisms, or tools, enable the installation to evaluate system performance and overall resource use. These mechanisms produce reports the installation can use to adjust MVS/XA in order to maximize its efficiency and, as a result, improve its job processing capability.

The remainder of this chapter describes these monitoring mechanisms. They are:

- The system management facilities (SMF)
- The Resource Measurement Facility (RMF) Version 3 (program product 5665-274)
- Dumping facilities, specifically SNAP dump, ABEND dump, SVC dump, stand-alone dump, and the dump reporting facilities: print dump and the interactive problem control system (IPCS)
- Trace facilities, specifically system trace, generalized trace facility (GTF), and master trace
- Serviceability level indication processing (SLIP)
- SYS1.LOGREC error recording

System Management Facilities

The system management facilities (SMF) collect information about MVS/XA processing that the installation can use to account for system use and to analyze system performance. SMF receives this information from various system services in the form of SMF records or obtains the information from various control blocks and builds SMF records. It writes these records to SMF data sets. These records describe system events, such as the start of TSO, the logon and logoff of TSO users, the reconfiguration of hardware, and individual job starts and terminations. SMF records also describe system status information, such as data set status (opened, closed, or scratched), VSAM catalog information, and job output statistics (cards punched and lines printed). An installation uses this recorded data to measure its processing capabilities, charge its users for processing time and resource usage, and make adjustments where necessary to provide better overall service. Usually SMF begins processing when MVS/XA starts, but,if necessary, the installation can stop or start SMF processing while MVS/XA runs.

Figure 9-1 presents an overview of SMF processing. The major elements of SMF processing are as follows:

1. SMF is part of the MVS/XA control program that runs in its own address space. It is initialized along with MVS/XA using the SMFPRMxx member of SYS1.PARMLIB which contains the parameters that define how SMF is to operate. Some SMFPRMxx parameters are required. Others are optional. Required parameters specify, for example, the identifier of the system on which SMF is running. Optional parameters specify, for example, the record types the installation chooses to have SMF write, whether the operator can modify SMF parameters, and whether SMF exit routines are to be used.

2. Various MVS/XA components include routines that provide data to SMF. Some components provide this data in complete records ready to be written to the SMF data sets; other components provide unformatted data, which SMF formats into records.

3. Some system routines that provide data to SMF also have interfaces with installation-written SMF exit routines to perform additional processing for certain events. The system routines invoke these exits at various times during job and job step processing.

4. Installation exit routines can, for example, enforce those standards of job processing unique to the installation (such as supplying defaults for missing JCL parameters), collect installation-dependent job information, or enforce the installation's standards for resource usage.

5. SMF routines collect unformatted data and format this data into SMF records, transfer records from the SMF buffer to the SMF data set, and issue messages to the operator indicating the successful or unsuccessful completion of specific SMF-related events.

6. SMF writes records to an SMF data set. When the data set is full, SMF writes records to another data set. The data in the full data set can then be saved on tape.

7. The installation can write analysis and report routines and use these to process SMF data. These routines execute as ordinary jobs. The analysis routines can collect SMF measurements into meaningful units of information by extracting or sorting the data and analyzing it. The report routines can format and print the results of the analysis. Reports on direct access volume activity, data set activity, and resource use can help an installation assess its computing efficiency.

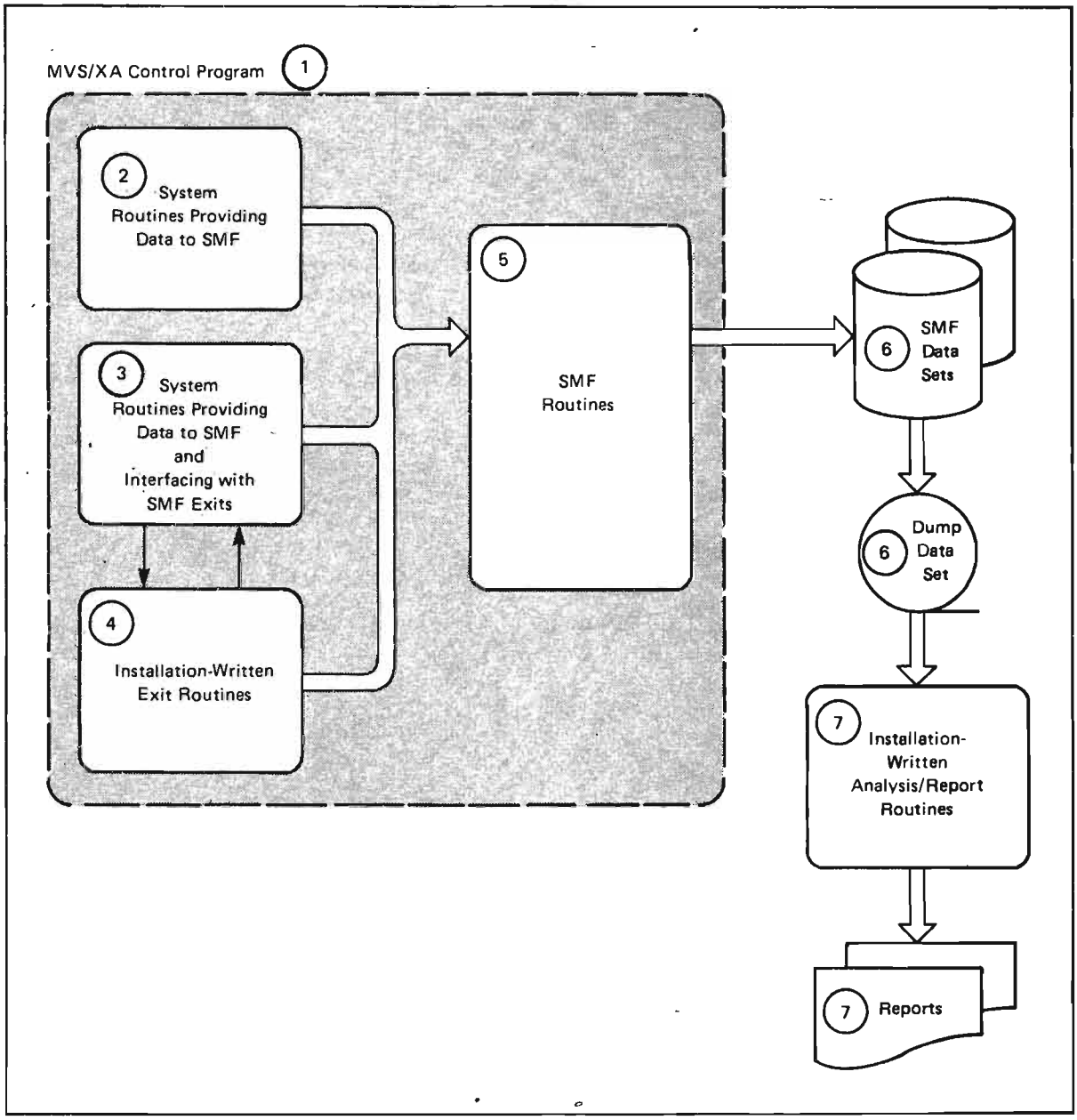


Figure 9-1. System Management Facilities - Overview

Resource Measurement Facility

The Resource Measurement Facility (RMF) Version 3 (program product 5665-274) is a measurement program the installation can use to analyze the performance of its system. RMF measures the use of many system resources, such as the processors, channel paths, devices, and real storage. Also, RMF measures the resource contention that enqueueing causes, the processing service that the system gives to different classes of users, the workflow (or speed) at which users move through the system, and the interaction that takes place among real storage, the processor, and the system resources manager (SRM).

An execution of RMF is called a **session**. Some sessions are of long duration, while others can be short. Some sessions are interactive, while others can be background jobs. The installation selects the sessions that best meet its needs.

Within a time interval, RMF measures data by exact count or by sampling. RMF makes an **exact count** measurement of a system indicator by computing the difference between its value at the beginning of an interval and its value at the end of the interval. RMF makes a **sampling** measurement of a system indicator by recording its value at each cycle within the interval; a **cycle** is a subdivision of an interval. (For example, each minute in an interval can be divided into sixty cycles that are one second long.) At the end of the interval, RMF gathers the data collected at each cycle and prepares to report the results. The installation controls the length of the interval and the cycle for the session.

Monitor I sessions measure a variety of system data over many intervals of time; they generally produce printed reports spanning large amounts of processing time. When each interval elapses, RMF summarizes the data it has measured, formats it, and reports it in a form the installation has selected.

Monitor II sessions, in contrast, take snapshots of the system's performance and produce either printed reports or reports on the screen for immediate inspection. Interactive sessions (called display sessions) can be short in duration; the interval of measurement is normally the time between two successive commands at the terminal.

Monitor III sessions collect information about the activities of users (units of work) and the delays they encounter when accessing system resources. Monitor III also measures the workflow of users and resources, which reflects the speed at which work moves through the system. Monitor III sessions are always interactive.

Through Monitor I, Monitor II, and Monitor III sessions, RMF can measure resource use in various system areas:

- Processor activity indicates the extent of wait time each processor experiences.
- Address space activity describes the status of address spaces and how they're being used.
- Channel path activity, I/O queuing activity and I/O device activity describe the use of the system's I/O configuration.
- Paging activity shows the amount of paging and swapping taking place.
- Workload activity shows what system services are being provided to particular users or groups of users.
- Page/swap data set activity describes the use of the paging data sets and swap data sets.
- ASM/RSM/SRM trace activity traces the contents of various control block fields that ASM, RSM, and SRM use to perform swapping for address spaces.
- Enqueue activity shows the contention for serially-reusable resources.
- Real storage/processor/SRM activity gives an overview of system activity.

- Virtual storage activity describes the use of common storage and private area storage.

- Transaction activity gives an overview of transaction activity by performance group period.

- Domain activity provides information on current domain definitions. A **domain** is a group of resources defined by the installation.

- Workflow indicates how jobs use system resources, the speed with which jobs move through the system, and how efficiently resources are serving job requests.

- Delay shows when a job is not productive because of contention for, or the unavailability of, some resource in the system.

The installation uses these measurements of system activity to identify use of system components and resources, to relate how well service is provided to different classes of users, to identify bottlenecks where contention for resources is high, and to locate excessive users of particular resources. Special RMF exception reports show when system performance reaches pre-selected thresholds.

RMF produces three forms of output: SMF records, printed reports, and display reports. The type of output RMF can produce depends on the type of RMF session. Monitor I and Monitor II sessions can produce SMF records for all activities measured. RMF can print reports either as a part of session processing or at a later time as part of post processing. The post processor can produce printed interval reports and various types of summary reports.

During Monitor III sessions, RMF produces screen displays of workflow and delay measurements rather than SMF records or printed reports. However, copies of individual screens can be printed.

The user starts an RMF session by issuing a **START RMF** command at a system console or by issuing an **RMFWDM** or **RMFMON** TSO command at a TSO terminal. During a non-display RMF session, the installation can use the **MODIFY** command to control RMF processing and display RMF status. An RMF session ends when its time limit expires or when the operator or terminal user stops the session.

RMF can invoke user exit routines at various points within a session; the type of session dictates the number of exits available. An installation exit routine, for example, can sample additional data at each cycle within a measurement interval, format and write its own SMF records, and produce its own reports.

Figure 9-2 summarizes the functions RMF provides.

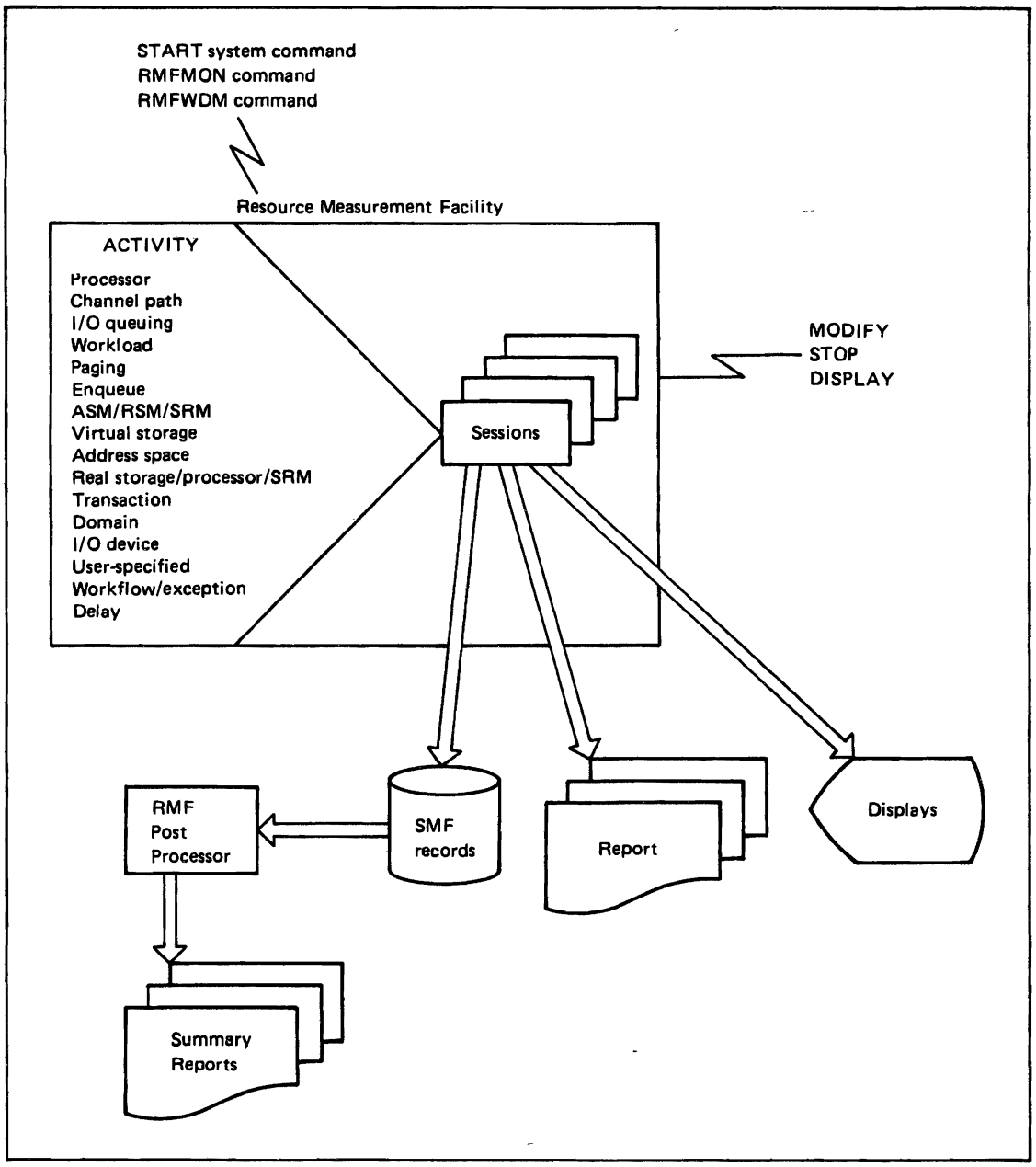


Figure 9-2. Summary of RMF

Dumping Facilities

Dumps are snapshots of what virtual storage looks like at a given instant in time; they are hard copy listing of the contents of the system's virtual storage locations. Dumps can include large areas of virtual storage or only a few locations. They can contain control blocks and data areas used by programs, the programs themselves, or both. While dumps can be taken to validate specific processing when the system is running normally, they are most frequently used to solve system problems and error conditions.

Dumping system information when an error occurs requires precise timing. As the system operates, the control blocks and data areas for both system and user

programs keep changing. Because these control blocks and data areas are volatile, taking a dump too early can reveal too little about a problem, and taking a dump too late can mean that the pertinent information has been overlaid with new data. A useful dump, then, is one that captures the contents of virtual storage when the error occurs or as close as possible to when the error occurs. Being able to take this kind of dump depends, to a large degree, on whether the error is job-related or system-related.

Job-related errors are those that a job can try to anticipate. That is, the user program or programs that make up the job include logic that plans for the occurrence of an error, such as an erroneous value in a control block or an unsuccessful return code from a called routine. When such a job-related error occurs, the program can immediately dump critical control blocks and data areas. These dumps then represent an accurate view of the contents of virtual storage that the problem solver can use to solve the problem.

System-related errors, on the other hand, are those that cannot be anticipated by a user job. A system-related error can affect the system, a major subsystem like the the job entry subsystem (JES) or the information management system (IMS), or several components of MVS/XA. This type of error is generally not localized to a specific job — although a specific job might be running at the time — and what to dump is not obvious. The MVS/XA dumping service, itself, might fail because of the system error. And, unless system activity is reduced shortly after such an error occurs, too much system information can change, rendering a dump of the error less useful.

MVS/XA dumping facilities handle either kind of error; the dumps they produce are the SNAP dump, ABEND dump, SVC dump, and stand-alone dump. SNAP and ABEND dumps are generally taken for job-related errors. SVC dumps and stand-alone dumps are generally taken for system-related errors.

Each of these dumps can contain two types of information: system data and program data. **System data** includes the nucleus, system queue area (SQA), local system queue area (LSQA), and control blocks associated with the units of work in MVS/XA, such as the TCBs, ASCBs, and SRBs. **Program data** includes the program's PSW, its register contents, its TCB and associated RBs, its save areas, and the program itself.

The remainder of this section presents more information about each of these dumps, including when they're used, how they're taken, and what output they produce.

SNAP Dump

The SNAP dump, as its name implies, is a snapshot of virtual storage requested by a program. This dump is formatted and easy to read. A program can take a SNAP dump at any time during its processing. During program testing, for example, a program can take a SNAP dump to print intermediate results of certain calculations. The programmer can analyze this dump to ensure that the program is operating correctly. For a job-related error, a program can take a SNAP dump to capture critical program storage areas at the time it detects the error. The programmer can then analyze this dump to determine the specific nature of the error and the reason for it.

To take a SNAP dump, the program uses the **SNAP** macro instruction; its operands identify the information to be dumped and the output data set for the dump. The

output data set can be sent to a printer for analysis of hard copy results, to a disk or tape for printing and analysis at a later time, or to a display terminal for viewing on the screen.

After the SNAP dumping service finishes processing the dump, it returns control to the program that invoked it. The program can then take other SNAP dumps at other points in its processing; the result is a comprehensive collection of information.

ABEND Dump

An ABEND dump is a display of virtual storage that a program can request directly when it can't circumvent an error and wants to terminate its processing. MVS/XA can also provide an ABEND dump indirectly when it detects job-related processing errors that can be circumvented by terminating the job. In either case, the program can't circumvent the error, and the only remaining action is to dump critical program storage and terminate. The programmer can then analyze this dump to determine what caused the abnormal termination.

To request an ABEND dump, the program uses the **ABEND** macro instruction with the DUMP operand. The ABEND dumping service writes the dump to a data set identified by a DD statement in the terminating job's JCL. This DD statement must be named SYSUDUMP, SYSABEND, or SYSMDUMP.

A SYS1.PARMLIB member exists for each of these names. Each member defines default dump options, which specify the default system and program data to be dumped to the SYSUDUMP, SYSABEND, or SYSMDUMP data set. The types of information dumped to these data sets are:

- **SYSUDUMP** - Storage associated with the failing task, such as its enqueue control blocks. This information is formatted by the ABEND dumping service and is ready for printing.

- **SYSABEND** - Storage associated with the failing task (same as the storage for the SYSUDUMP DD statement) with the addition of the local system queue area and IOS control blocks. This information is formatted by the ABEND dumping service and is ready for printing.

- **SYSMDUMP** - Storage used by the system to process the failing task, such as the nucleus, the system queue area, the local system queue area, the scheduler work area, and the private area of the address space for the failing task. This information is not formatted by the ABEND dumping service. Analysis and formatting programs can process this output to produce a readable dump. The PRDMP MVS/XA service aid (called print dump) and the interactive problem control system (IPCS) are such programs. IPCS allows the programmer to format and view dumps at a display terminal without having to print them. The dump analysis and elimination (DAE) function suppresses duplicate dumps that come from the same problem.

The program requesting the ABEND dump can accept the default dump options or alter them through other operands on the ABEND macro. The final contents of the ABEND dump, however, might not be what the program requested because the operator can alter the system default dump options through the CHNGDUMP command. Also, any recovery routines (invoked by the recovery termination manager as a result of the program's abnormal termination) can alter these dump options.

SVC Dump

SVC dumps serve system programs in the same way as SNAP dumps serve user programs. That is, SVC dumps are the control program's equivalent to the user program's SNAP dump. Also, only authorized programs or those running in control program key can request SVC dumps. Among these programs are:

- **Programs that are part of MVS/XA.** These programs take SVC dumps for system-related errors they can anticipate.
- **MVS/XA recovery routines (FRRs and ESTAEs).** These programs take SVC dumps for unanticipated system-related errors that occur in the programs that define them.
- **Authorized installation programs and user modifications to MVS/XA.** These programs can take SVC dumps both for system-related error conditions and as part of normal processing. SVC dumps during normal processing help to test the program before installing it in MVS/XA.
- **Programs that process the DUMP operator command.** The operator issues the DUMP command for certain system error conditions, and these programs request the SVC dump.

To take an SVC dump, the program issues the **SDUMP** macro instruction, either specifying operands that identify the information that is to be dumped and a specific data set to be used for the dump or accepting the system default options. As with ABEND dumps, the operator can change the default SVC dump options through the CHNGDUMP command.

SVC dump output data sets (named SYS1.DUMPxx) reside on disk or tape. Because SVC dump output is unformatted on these data sets, an analysis and formatting program must process this dump output to produce readable dumps. Similar to the ABEND dump for the SYSMDUMP DD statement, the PRDMP MVS/XA service aid and the interactive problem control program (IPCS) can be used to format the SVC dump into a readable form. IPCS allows the programmer to format and view dumps at a terminal without having to print them.

After the SVC dump service finishes producing the dump, it returns control to the program that invoked it. The program can then take additional SVC dumps at other points in its processing. This ability to take several SVC dumps is helpful for any recovery routine that handles system-related errors. By requesting SVC dumps at various points in its recovery processing, the recovery routine can produce a comprehensive collection of system program information reflecting its recovery actions. If the system does fail, these SVC dumps can help to isolate the cause of the failure. To avoid unneeded SVC dumps (those dumps of the same problem), the dump analysis and elimination (DAE) function is used to suppress duplicate dumps.

Stand-Alone Dump

A stand-alone dump is a dump produced by a program that the operator executes. When MVS/XA fails, the operator loads the stand-alone dump program into storage from a volume where it resides. The program runs by itself and dumps all of real storage and selected portions of virtual storage. The dump includes the nucleus, the trace table of system events, the real storage contents and selected

virtual storage contents of all address spaces, the prefixed storage area (PSA), and the system queue area (SQA).

The first step in running stand-alone dump program is to activate the store status procedure that stores the processor time, current PSW, general purpose registers, and other processor-type information into permanently assigned locations in storage. The store status procedure can be invoked by an operator command or by an option of the stand-alone dump program. This procedure preserves in the dump the processor status existing at the time the system failure was detected.

There are two forms of stand-alone dumps: a low-speed form and a high-speed form. The low-speed form dumps storage and automatically formats it for printing. The high-speed form merely dumps storage in unformatted form to tape or disk for formatting and printing at a later time. (PRDMP and IPCS can be used to format the high-speed stand-alone dump).

Figure 9-3 summarizes the MVS dumping facilities.

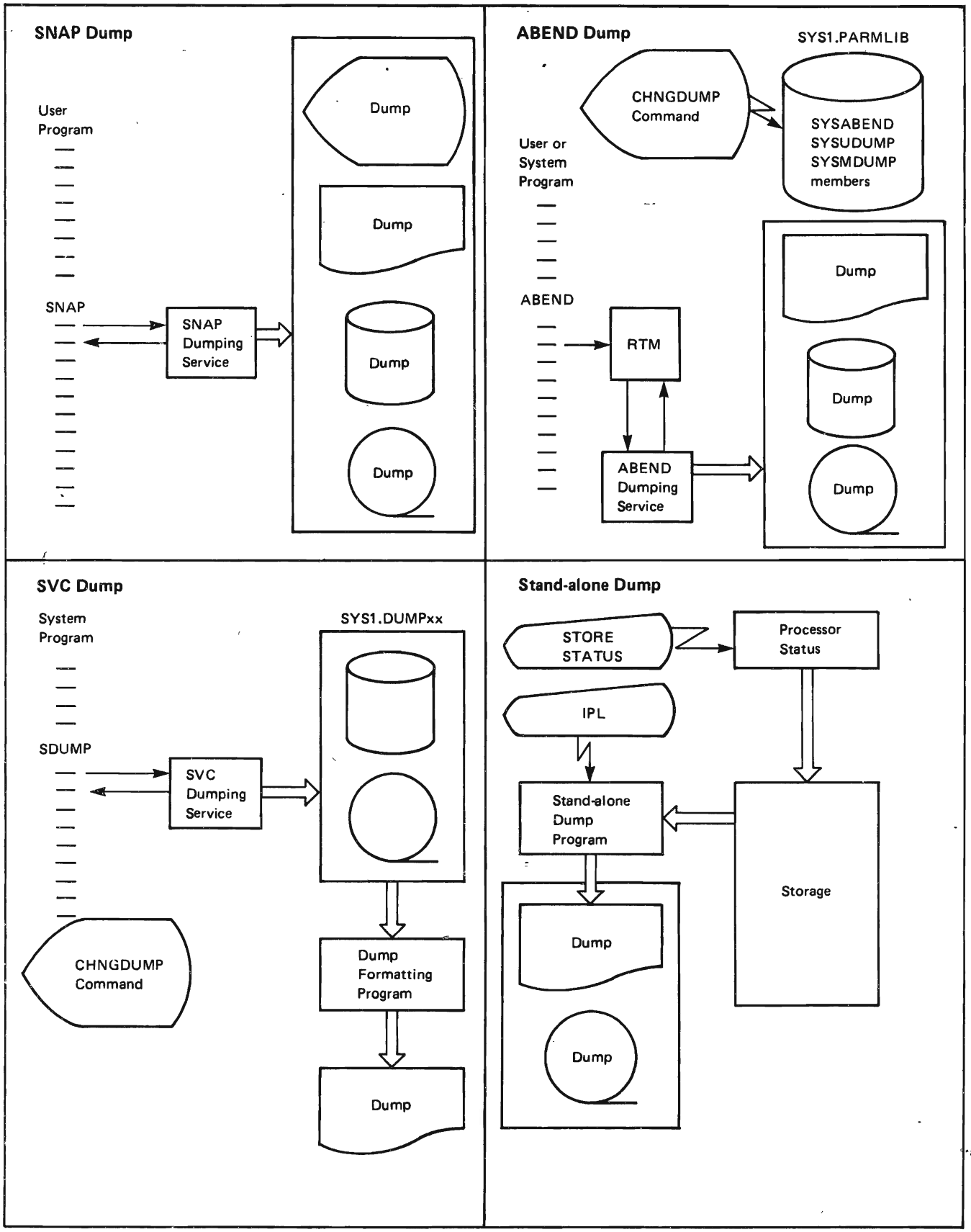


Figure 9-3. Summary of MVS/XA Dumping Services

Trace Facilities

Tracing system events provides valuable information for performance analysis and problem diagnosis. For example, a sequence of I/O interruptions from specific devices can pinpoint high or low device use. Or, a sequence of program interruptions can either eliminate programs as possible sources of an error or, in fact, isolate the program that did cause the error. Any tracing mechanism must be able both to capture the system event and to record pertinent information about the event for later use.

There are three MVS/XA trace facilities: **system trace, master trace** and **generalized trace facility (GTF)**. System trace and GTF record the same internal system events although GTF does so in more detail.

System trace is most useful when an unexpected problem occurs and GTF is most useful when there is a known problem and a need to narrow it down. GTF uses more system resources than system trace.

Master trace records external system activity such as the commands entered by the operator, responses to these commands, and other system messages.

Aside from these three trace facilities which record system events, each installation has either JES2 or JES3 traces to monitor job entry and output processing.

Installation-written programs, MVS/XA system components, and first-level interruption handlers activate the MVS/XA trace facilities by a mechanism known as hook processing. A **hook** is a sequence of instructions that signal to the trace facility that the event has occurred, capture the relevant system data, and either pass this data to the trace facility directly or store the data until the tracing mechanism records it. Figure 9-4 shows how a hook can capture information about a program interruption. The sequence of events is as follows:

1. Program A attempts to store data in an area of storage that is protected from access. This action causes a protection exception program interruption.

2. When the program interruption occurs, the processor immediately switches control from program A to the program check first-level interruption handler (PCFLIH), which saves the processing status of the interrupted program.

3. The PCFLIH contains a hook, a sequence of instructions that calls the tracing mechanism to trace the program interruption.

4. The tracing mechanism, after recording the program interruption event and the relevant system data (such as processor identifier, time of the interruption, PSW of program A) preserved by the PCFLIH, returns to the PCFLIH, which finishes its processing of the interruption.

Each MVS/XA tracing mechanism is started and stopped by operator commands. The operator uses the **TRACE** command with the ON or OFF operand to start or stop system trace and master trace. The START GTF and STOP GTF commands perform the same functions for GTF. All tracing mechanisms can be on at the same time.

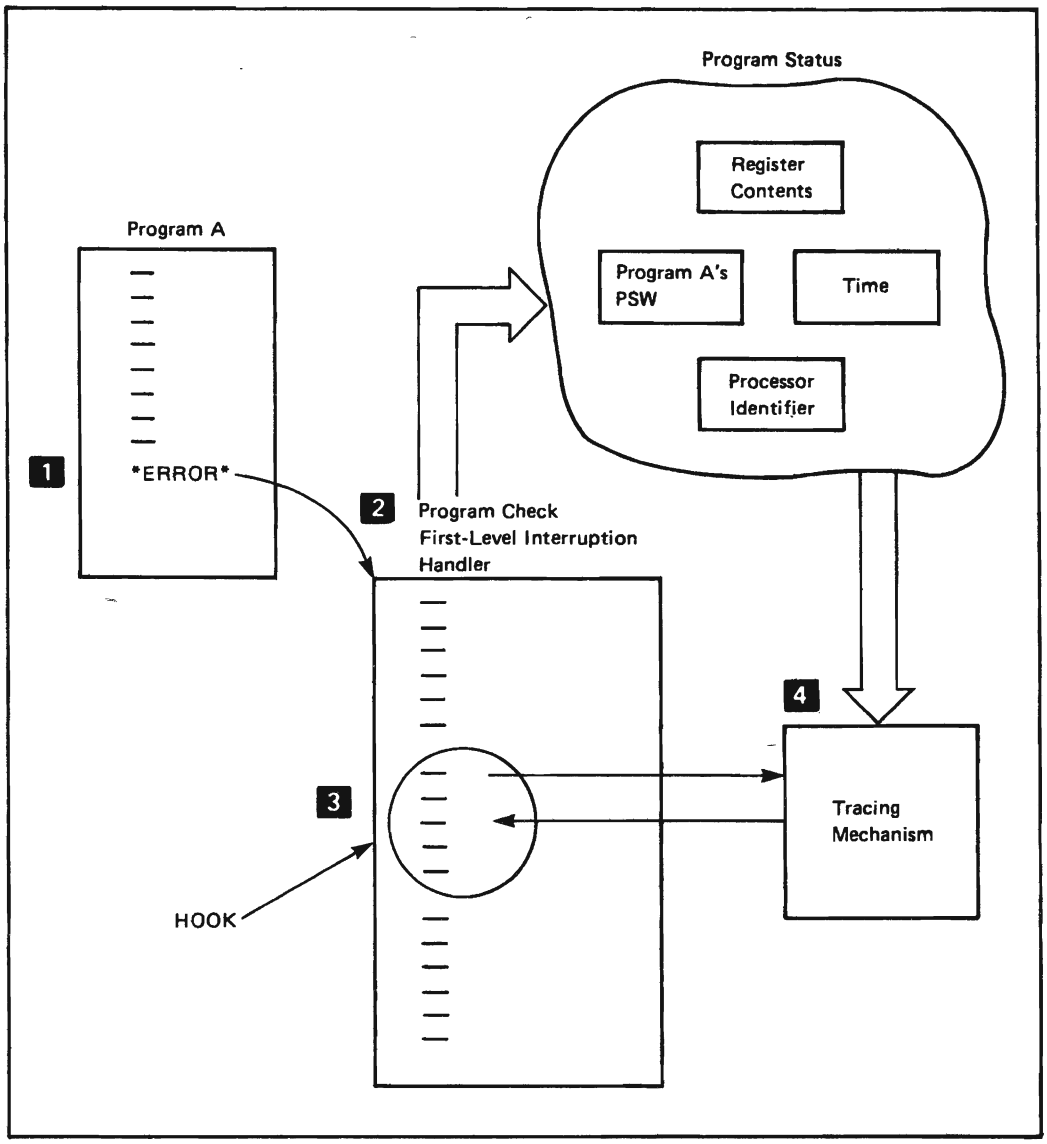


Figure 9-4. The HOOK Concept

System Trace

The installation can start or stop system trace and control trace options by means of the TRACE command. Usually, the installation starts system trace when it starts MVS/XA and, from that point on, system trace records the following types of events:

- Start subchannel
- Modify subchannel
- Halt subchannel
- Clear subchannel
- Resume subchannel
- External interruption
- Emergency signal (EMS) external interruptions
- Service signal external interruption
- External call external interruption
- Clock comparator external interruption

- SVC interruption
- SVC return
- SVC error
- Program interruption
- I/O interruption
- Task dispatch
- Initial SRB dispatch
- Suspended SRB dispatch
- Wait task dispatch
- Machine check interruption
- Restart interruption
- Alternate CPU recovery
- Lock Suspension
- Trace options alteration
- User event trace
- Program Call (PC) control instruction
- Program Transfer (PT) control instruction
- Set Secondary Address Space Number (SSAR) control instruction
- Branch and Link (BALR) general instruction
- Branch and Save and Set Mode (BASSM) general instruction
- Branch and Save (BASR) general instruction

System trace logs each event in a system trace table in the TRACE address space in virtual storage. The system trace table consists of a queue of buffers for each processor running under one MVS/XA system. The installation can control the size of the trace table with the TRACE command. Each entry in the table includes the following information for each event:

1. A unique code that identifies the event

2. Data associated with the program affected by the event, such as the processor identifier, the address of the current TCB, and the contents of the PSW

Figure 9-5 illustrates the system trace function and the MVS/XA components that invoke it.

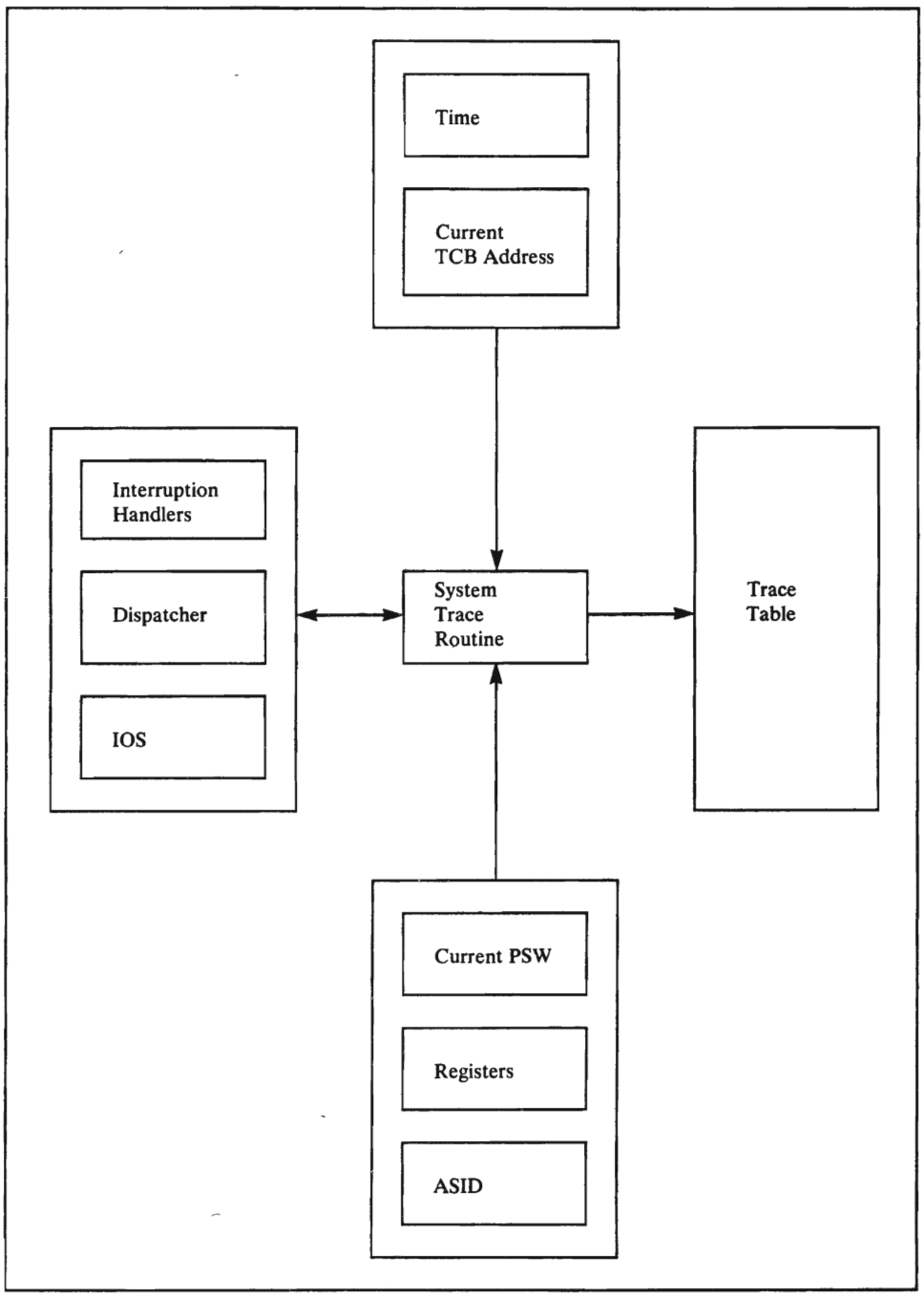


Figure 9-5. System Trace Overview

Stand-alone dumps always contain the system trace tables, and users can request that the system trace tables be copied into their ABEND dumps, SNAP dumps and SVC dumps. The print dump (PRDMP) service aid utility program includes a TRACE verb to request formatting and printing of the trace table entries.

Generalized Trace Facility

The generalized trace facility (GTF) provides greater event and data selectivity than system trace and produces trace output in more ways. GTF traces most of the same events as system trace, and also traces events such as when the recovery termination manager routes control to recovery routines (FRRs and ESTAEs),

when MVS/XA programs invoke the system resources manager, and the processing activities associated with a VTAM network and subsystems.

GTF traces in two modes: internal mode or external mode. In **internal mode**, GTF builds the trace records in virtual storage. Users of the MVS/XA dumping facilities can optionally request that these records be incorporated into their ABEND dumps, SNAP dumps, and SVC dumps. (Stand-alone dumps always contain GTF trace records.) In **external mode**, GTF provides the same function as for internal mode but also writes each trace record to a data set that resides on an external storage device (either a tape or disk). The trace records on the external storage device can be formatted, analyzed, and printed at a later time to produce reports of system activity. The EDIT function of the PRDMP MVS/XA service aid program is normally used to format these GTF trace records.

GTF is a started task. The system operator issues the **START** command to start GTF and the **STOP** command to stop it. The options that govern its operation reside in the GTFPARM member of SYS1.PARMLIB; these options define the events GTF is to trace and the mode of tracing GTF is to use. The operator has the ability to override these options.

Like system trace, GTF uses hooks to trace system events. The difference between system trace hooks and GTF hooks, though, lies in how the hook causes tracing to occur. System trace hooks invoke system trace directly, in contrast to GTF hooks, which cause a program interruption that switches control to GTF. The **monitor call (MC) instruction**, which is part of each GTF hook, selectively produces this program interruption.

GTF uses this characteristic of the MC instruction to define classes of events that it can monitor. When GTF is started, these classes of events are specified as trace options in GTFPARM or as responses to GTF prompt messages. Hooks for events that match the initialized events cause the MC program interruption and switch of control to GTF. Hooks for events that do not match the initialized events cause no MC program interruptions; these events are ignored and not traced.

Programs use the HOOK or GTRACE macro instructions to set the hooks that trace the system events. MVS/XA supervisor functions use the HOOK macros to trace, for example, program interruptions, dispatches, and RTM routing to recovery routines. User programs and subsystems use the GTRACE macro to trace events unique to them.

Figure 9-6 summarizes GTF processing; the figure highlights the following processing steps:

1. The system operator starts or stops GTF at the system console using the START and STOP commands. The GTFPARM member of SYS1.PARMLIB or operator replies to GTF prompting messages define the system events GTF is to trace.

2. GTF operates in internal mode or external mode. In internal mode, GTF builds the trace records in storage. In external mode, GTF builds the trace records in storage and also writes the trace records to a data set for printing or analysis at a later time. MVS/XA dumping facilities can include trace records in dumps regardless of whether GTF is operating in internal or external mode.

3. User programs or subsystems use the GTRACE macro to define GTF hooks to record events unique to them. Supervisor programs use the HOOK macro to define GTF hooks to record system events.

   These macros generate monitor call instructions that cause a program interruption if the event defined by the hook matches an event in GTFPARM. As a result of the program interruption, the processor switches control to GTF to trace the event defined by the hook. After GTF traces the event, control returns to the program that invoked GTF.

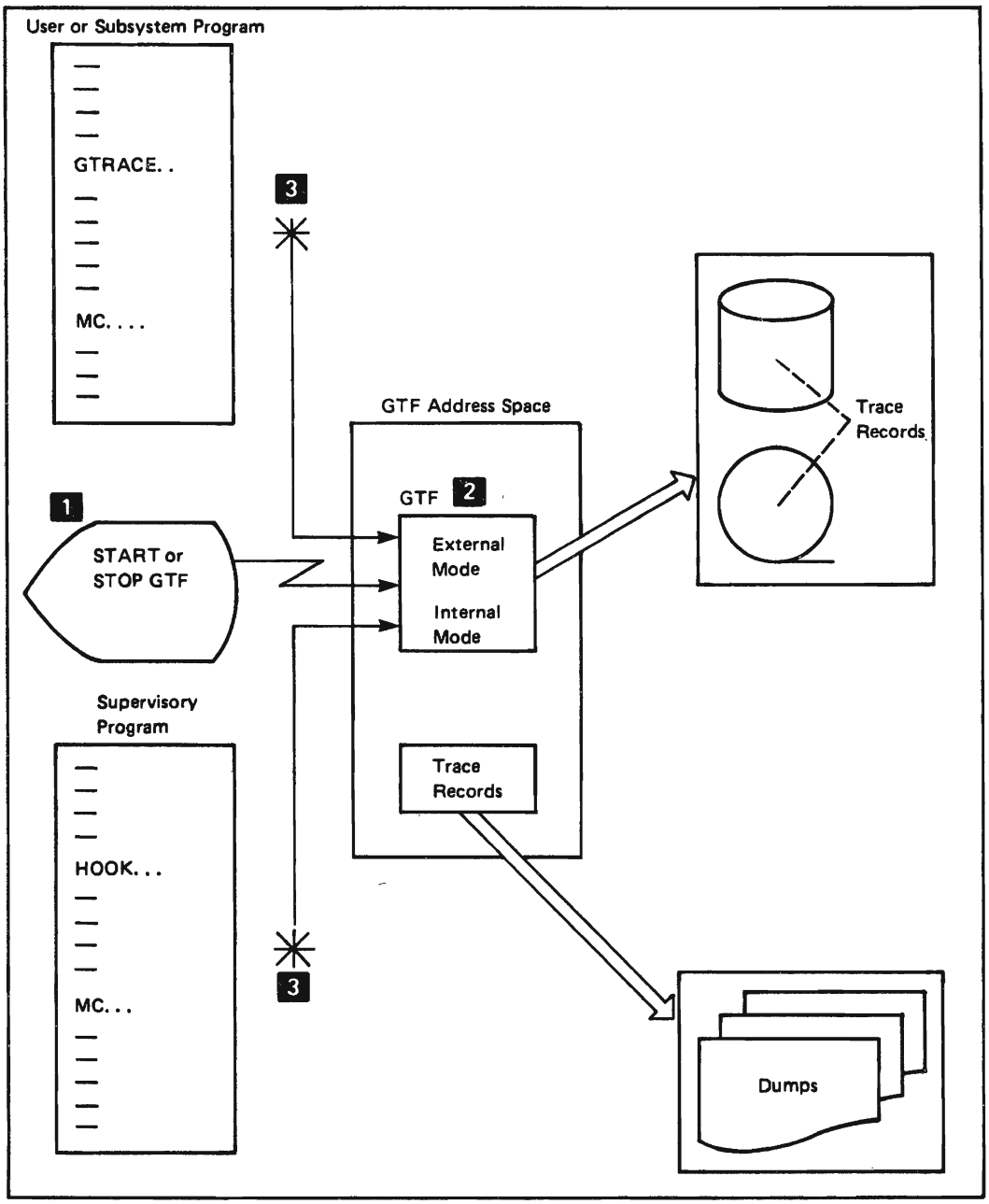


Figure 9-6. Generalized Trace Facility - Summary

Master Trace

MVS/XA records console traffic through master trace. Unlike the tracing functions of system trace and GTF, which preserve internal system activity (I/O interruptions, dispatches, routing to FRRs, and so forth), master trace preserves external system activity, such as mount messages, status displays, operator-issued commands, system responses to commands, and other messages, recording this activity when it occurs in a table in storage.

When the master trace function is started, the communications task schedules the tracing. Because the communications task normally handles message traffic within MVS/XA anyway, it is in a perfect position to trace such traffic; it routes each message to master trace, and master trace preserves each message in the master trace table. (A hardcopy log function, separate from master trace, provides a permanent record of the same kinds of console traffic that master trace preserves.)

A hard copy listing of the operator console message traffic can help in debugging a system failure, especially if an I/O device caused the failure. In reconstructing the events that led up to such a failure, the system trace table (previously described in this chapter) would contain entries for I/O errors that occurred when programs accessed the faulty device. This information might be enough to pinpoint a device problem, but a listing of console traffic showing when volumes were mounted would also help. By comparing the time when a volume was mounted to the times associated with the I/O errors, the problem solver can pinpoint the problem as a faulty device. Knowledge of console traffic, then, generally helps to create a more complete picture of the system environment and decreases the chance of overlooking obvious causes of errors.

Because the master trace table resides in virtual storage, it can also be dumped. That is, users of those MVS/XA dumping facilities, described earlier in this chapter, can request that the contents of the master trace table be included in their dumps, thus providing a more complete collection of information regarding an error condition. The installation sets the size of the master trace table with operands on the **TRACE** command issued to start master trace.

Figure 9-7 illustrates the master trace function.

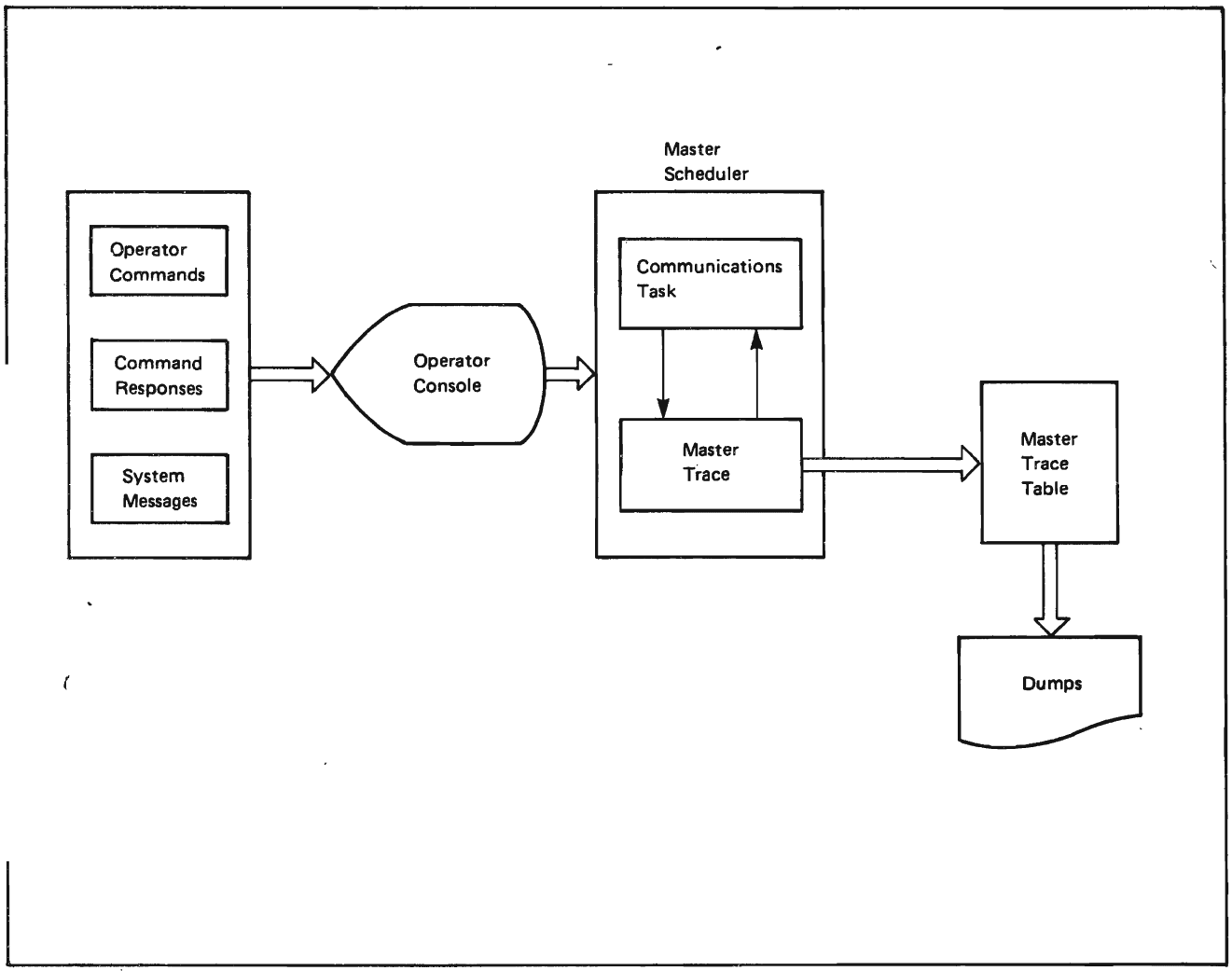


Figure 9-7. Master Trace Overview

Serviceability Level Indication Processing (SLIP)

Serviceability level indication processing (SLIP) aids in error diagnosis. Diagnosing a problem requires information about the problem. This information includes the events that led to the error and the contents of critical data areas and control blocks at the time of the error. Dumps supply a picture of virtual storage when the error occurs. Traces supply a record of system events. SLIP joins these two diagnostic mechanisms into a powerful debugging tool that associates a prescribed diagnostic action, like dumping or tracing, with a specific event, like a program interruption, ABEND, or storage reference.

The description of the system event that is to be intercepted and the action to be taken as a result is called a **trap**. At the operator console or an authorized TSO terminal, the problem solver enters the **SLIP** command to describe each trap. Operands on the SLIP command specify the system event to be intercepted, the action to take place when the event occurs, and whether the trap is to be enabled or disabled. An **enabled** trap is one for which the action is taken if the system event to be intercepted does, in fact, occur. A **disabled** trap, on the other hand, is ignored; that is, no check is made for the system event. The problem solver can enable and disable traps as system conditions change.

SLIP traps can intercept two classes of system events: program event recording (PER) events and error events.

Program Event Recording Events

PER events take place because the processor can cause a program interruption when certain system events occur. Specifically, the PSW, which controls the processor's execution of instructions, contains a program event recording bit. When this bit and the bits in control register 9 that correspond to a particular condition are on, a program interruption or PER interruption (as it is commonly called) occurs. This may be for one of the following conditions:

- The instruction executed was fetched from a storage location that falls within a specific range of addresses.
- The instruction executed is a successful branch instruction.
- The altered storage location falls within a specific range of addresses.

The PER interruption that occurs in these cases is handled by the program check first-level interruption handler (PCFLIH), which alters the sequence of processing from the program that contains the instruction to SLIP. The processor, in effect, recognizes an instruction fetch, a successful branch, or a storage alteration of a program and gives control to SLIP. After SLIP processes the PER event, it normally returns control to the interrupted program, (although SLIP traps can be defined so that the program interrupted by PER events is abnormally terminated).

Error Events

Error events are a subset of errors that cause recovery termination management (RTM) processing. Chapter 10, "Recovering From Errors" lists the errors that cause RTM processing. Some of the errors that SLIP can trap are:

- Program check interruptions. Programs cause errors, such as an addressing exception and a storage protection check.
- Dynamic address translation errors. The DAT hardware fails or the contents of the page tables become invalid.
- Machine checks. The machine check is not recoverable by the hardware, and the software must try to recover.
- Abnormal address space termination. MVS/XA components request RTM to terminate an address space and clean up its resources.
- ABEND. A task abnormally terminates.
- SVC error. A locked, disabled, or SRB mode program issues a supervisor call instruction.
- Restart interruption. The operator presses the restart key on the system console.

SLIP Actions

For either a PER event or an error event, SLIP can perform one of the following actions:

- Take an SVC dump tailored to the needs of the problem solver.
- Cause a GTF trace record to be written.
- Put the system into the wait state so that the problem solver can manually display or alter storage or take a stand-alone dump.
- Ignore the event altogether.
- Override the suppression of a dump by the dump analysis and elimination (DAE) facility.

For error events, SLIP can also suppress selected dumps.

Using SLIP Traps

The SLIP command can define a trap, alter the state of existing traps (that is, enable or disable them) to meet new system conditions, or delete traps that are no longer useful. SLIP traps can be defined so that they are automatically disabled after they have been matched a specified number of times. Also, SLIP traps for PER events can be defined so that they are automatically disabled when processing the PER events identified by these traps consumes a specified percentage of system processing time.

The system interprets a sequence of SLIP traps in a "last-in-first-out" (LIFO) order. That is, the most recently-defined trap is processed first, then the next most recent, and so on until the conditions specified in the trap match the system events. When the match occurs, SLIP takes the action specified by the trap, and the process of interpreting the traps begins again in LIFO order with the most recently-defined trap. The problem solver uses this ordered processing of traps to control the way in which system events are intercepted. For example, assume that a program is modifying location X, but JES2 is the only program that should modify location X. To identify any other program that is modifying location X, the problem solver sets two traps in the following order:

1. TRAP1: A SLIP trap to intercept the PER event of storage alteration for location X for all programs in all address spaces. Take an SVC dump if this event is intercepted.

2. TRAP2: A SLIP trap to intercept the PER event of storage alteration for location X for only address space 2, which belongs to JES2. Ignore the event when it's intercepted.

Because of the LIFO order in processing traps, TRAP2 is processed first. When JES2 alters location X, the event is ignored. TRAP1 is not processed. When a program running in an address space other than address space 2 alters location X, TRAP2 is processed but does not match. TRAP1 is then processed. TRAP1 does match this event, and an SVC dump is taken. In this way, a sequence of SLIP traps can be designed to filter out known processing and expose unknown processing.

Figure 9-8 summarizes the basic SLIP concepts and functions. The following description highlights these concepts and functions:

1. The problem solver establishes a SLIP trap by entering the SLIP command at the system console or an authorized TSO terminal.

2. A PER event SLIP trap interrupts the program when a PER event occurs. The PER events are instruction fetch, successful branch, and storage alteration.

3. An error event SLIP trap intercepts error events. SLIP error events are a subset of those errors that cause RTM processing. They include program check interruptions, SVC errors, and DAT errors.

4. Each SLIP trap indicates actions that are to take place when a PER event or error event is intercepted. These actions include dumping critical storage areas and control blocks, writing GTF trace records to the SYS1.TRACE data set, or ignoring the event altogether.

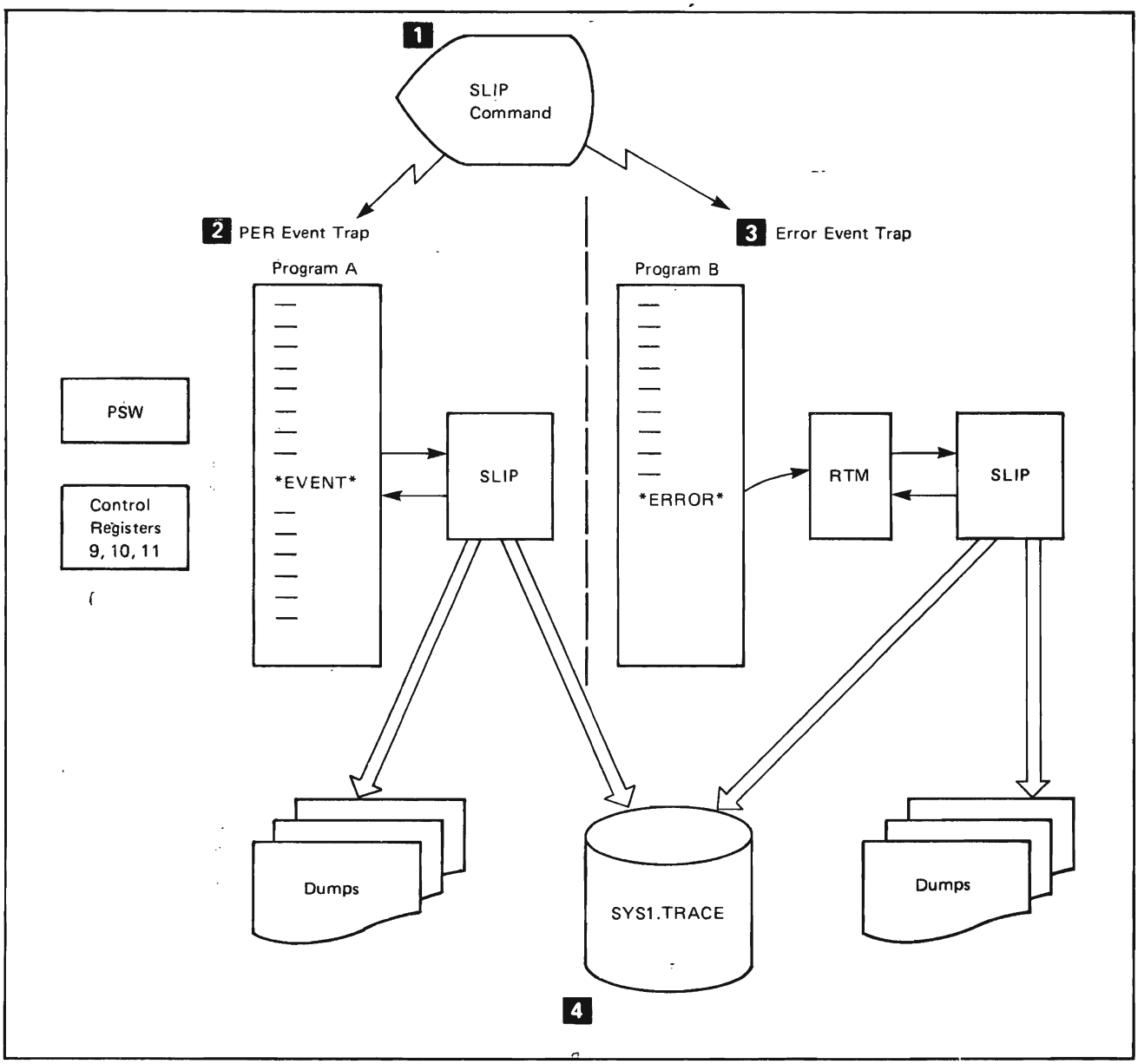


Figure 9-8. Serviceability Level Indication Processing Summary

SYS1.LOGREC Error Recording

Diagnosing errors in MVS/XA can require more information than is supplied by those monitoring and dumping mechanisms already described. In order to recreate certain environmental conditions important to the solution of the problem, the problem solver might need a knowledge of the system's complete history, sometimes going back as far as when the system was initialized for operation. The time when early system events occurred and the order in which they occurred can help to reveal the cause or causes of system failures.

SYS1.LOGREC error recording creates such a history by recording hardware failures, selected software errors, and other system events for the entire processing life of the system — from initialization to shutdown. Various MVS/XA control

programs write system error information to SYS1.LOGREC, a permanently-resident system data set, creating, over a period of time, a system history. The recovery termination manager (RTM), for example, records the error analysis that the machine check handler does for a machine check interruption and also records the data that error recovery routines supply about software errors.

SYS1.LOGREC is one of the data sets that can be on the system residence (SYSRES) volume or a user-specified volume. During the first stages of MVS/XA initialization, SYS1.LOGREC error recording begins; it ends only when the system stops operating (whether through normal shutdown or abnormal failure). The SYS1.LOGREC data set, then, becomes a running log of valuable information about errors -- such as hardware errors associated with failing storage or devices and software errors associated with failing programs -- that occurred during the system's operation. The installation can use this information to make configuration changes and debug system problems.

Figure 9-9 illustrates the following steps in SYS1.LOGREC error recording:

1. During system generation, the IFCDIP00 service aid program initializes the SYS1.LOGREC data set. This program creates a time stamp record that contains the time when MVS/XA was generated, the time of a forthcoming IPL, and various other system-related data; this record is the starting point for the history of MVS/XA processing. After IFCDIP00 finishes the initialization, SYS1.LOGREC is ready to receive error records.

2. During system operation, various MVS/XA routines format and write records to SYS1.LOGREC about failing hardware (such as a device or a processor), software errors (such as program errors, machine checks, ABENDS), and other system events (such as device demounts, reconfigurations, and end-of-day or shutdown events). For most of these situations, the recording routines write to SYS1.LOGREC regardless of whether or not the system was able to recover from the error. Each record, while identifying the error and the time it occurred, also contains other information, such as the current device hardware status, any results of software recovery, and statistical data on the number of such errors that have occurred to date.

3. The environmental recording, editing, and printing program, EREP, retrieves data from SYS1.LOGREC to (1) produce reports useful for diagnosing system errors or to (2) dump the SYS1.LOGREC data to an auxiliary data set so that SYS1.LOGREC can be used again. Many auxiliary data sets can be generated as SYS1.LOGREC fills up, forming an archive of SYS1.LOGREC data that the installation can use to extend the history of error activity beyond the capacity of SYS1.LOGREC. To produce detailed reports of the system's error activity, EREP can process any or all of the data sets in the archive, including the data on SYS1.LOGREC itself.

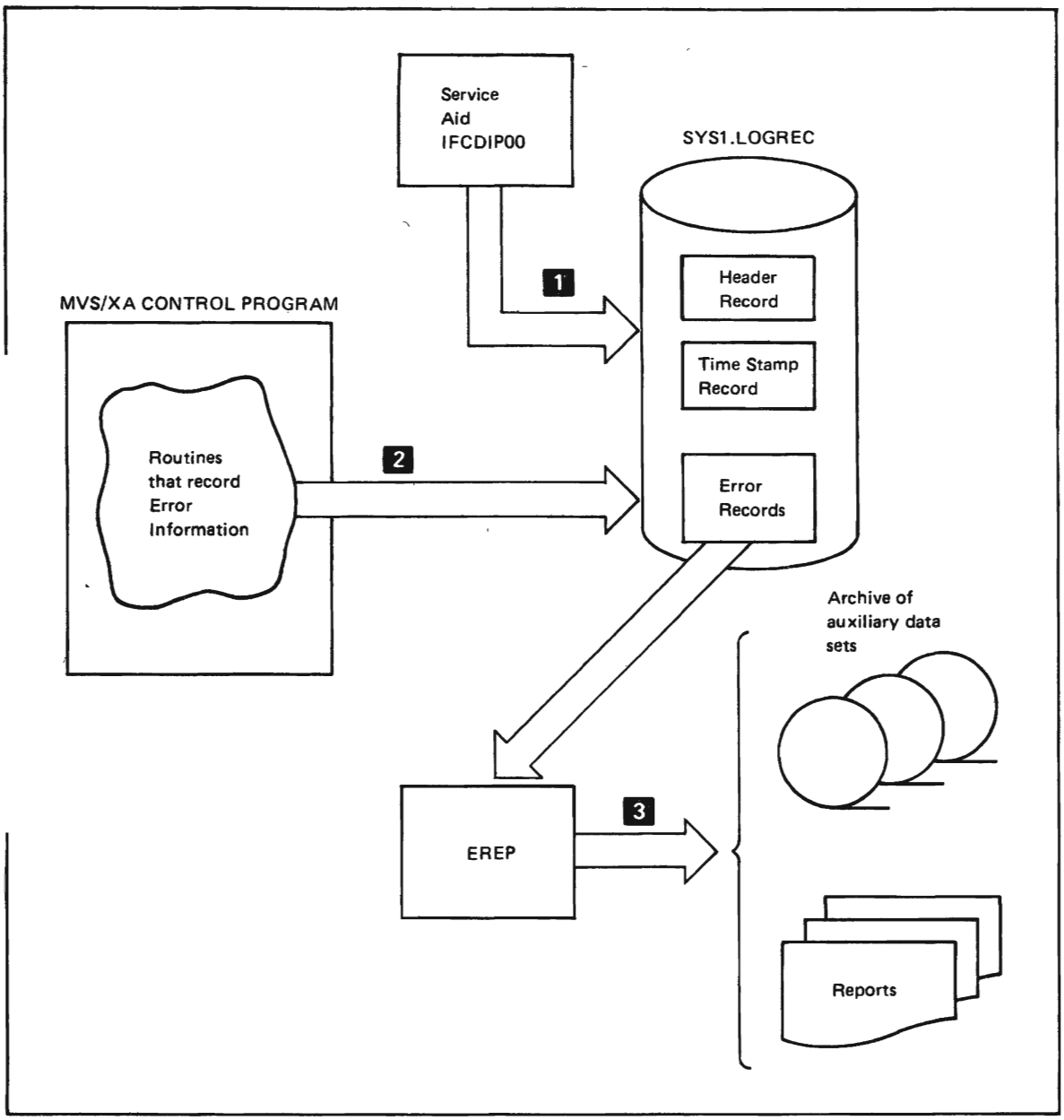


Figure 9-9. SYS1.LOGREC Error Recording Overview

Chapter 10. Recovering From Errors

A system is available when both its hardware and software are capable of processing jobs. Error recovery in MVS/XA is designed to increase the availability of the system and reduce the impact on users when errors occur in critical software and hardware components. If recovery is not possible, MVS/XA attempts to continue without the damaged facility. In general, recovery is attempted in such a manner that the recovery processes are transparent to the user.

Recovery routines have four objectives:

- To isolate the error
- To assess the damage and attempt to confine it to one user or task
- To indicate the actions, such as dumping, that should be taken
- To repair the damage and perform clean-up processing so that the function can be restarted

In MVS/XA, error processing of software failures is handled by recovery termination (RTM), and error processing of hardware failures is handled by several facilities. As a result, MVS/XA processing continues with minimal downtime.

Software Recovery: Recovery Termination Manager

The recovery termination manager (RTM) monitors the flow of software recovery processing by handling all abnormal termination of tasks and address spaces, and passing control to recovery routines associated with the terminating functions. The RTM enables user programs to establish their own recovery protection and system programs to enhance system serviceability and reliability.

The RTM is invoked for the following conditions:

- I/O error during a page-in operation
- Program error not handled by a program interruption routine
- Machine error not handled by hardware recovery
- Supervisor call that is invalid
- Restart operation initiated by the console operator
- CALLRTM macro instruction directed toward another task (ABTERM)
- CALLRTM macro instruction directed toward an address space (MEMTERM)
- ABEND macro instruction
- Dynamic address translation (DAT) error
- Branch entries for abnormal termination requests
- Reentry for abnormal termination requests
- Reentry for machine checks

Recovery Routines

Two types of recovery routines are identified by the RTM: task recovery routines and functional recovery routines. These routines are described in the following sections.

Task Recovery Routines

Task recovery routines - extended specify task abnormal exit **(ESTAE/ESTAI)** - provide recovery for those programs that run enabled, unlocked, and in task mode. They are established by using the ESTAE macro instruction or the ESTAI parameter of the ATTACH macro instruction.

A program can intercept an anticipated ABEND by issuing the ESTAE macro or the ATTACH macro with the ESTAI option. Control is given to a user-specified routine in which the user may perform pretermination processing, diagnose the cause of the abend, and specify a retry address if he wishes to avoid the termination. The routines operate in the mode (problem program or supervisor) that existed at the time the ESTAE request was made.

Note: STAE/STAI, specify task abnormal exit, are available with OS/VS2 Release 1 (SVS) and with OS/MVT and OS/MFT. Although STAE and STAI are also available in MVS/XA, it is recommended that ESTAE or ESTAI be used in MVS/XA. ESTAE or ESTAI provide increased capabilities over STAE or STAI; they can schedule clean-up processing under certain instances for which STAE routines do not get control and can provide defaults for the most commonly used options.

If a task is scheduled for abnormal termination, the recovery routine specified by the most recently issued ESTAE macro instruction gets control. If the ESTAE routine cannot provide recovery for the error, the next higher-level ESTAE routine (if any) associated with the task is gets control. This process of passing control from a recovery routine to a higher-level recovery routine along a pre-established path is called **percolation**.

Each ESTAE routine for the task is then given control, one at a time in LIFO (last-in first-out) order, until retry is requested or all routines for the task are exhausted. When ESTAE processing is exhausted, abnormal termination occurs.

Functional Recovery Routines

Functional recovery routines (FRRs) provide recovery for those system programs that run disabled, locked, or in SRB (service request block) mode, or less frequently, for programs in supervisor state, key 0, that run enabled and unlocked. The system programs establish the FRR by using the SETFRR macro instruction.

The SETFRR macro instruction provides each system program with the ability to define its own unique recovery environment. Each FRR established by a system program is placed in an FRR LIFO (last-in first-out) stack that is used during RTM processing. The SETFRR macro instruction can be used to add, delete, or replace FRRs in the stack.

Each FRR stack used by the RTM contains the addresses of the FRRs established to protect a single path through the system control program. When an error occurs in a path, the RTM passes control to the last FRR in the associated stack. If the FRR cannot provide recovery for the error, the previously-established FRR in the stack is given control (percolation). Each FRR in the stack is eventually given control, one at a time in LIFO order, until retry is requested or the stack is exhausted. When FRR processing is exhausted, appropriate task recovery routines (if any exist) are given control; otherwise, abnormal termination occurs.

Any user-written routines outside the control program that are qualified to issue the SETFRR macro instruction may add one, and only one, FRR to a stack. If more than one FRR is added to a stack, abnormal termination may occur when SETFRR is issued.

Hardware Recovery Facilities

MVS/XA facilities gather information about hardware reliability and allow retry of operations that fail because of processor, I/O device, or channel errors. The facilities are designed to keep the system operational in the event of hardware failures.

The hardware recovery facilities are:

- Machine check handler (MCH)
- Alternate CPU recovery (ACR)
- Subchannel logout handler (SLH)
- Dynamic device reconfiguration (DDR)
- Missing interruption handler (MIH)

Machine Check Handler

The machine check handler (MCH) minimizes the impact of machine malfunctions on MVS/XA systems. It alerts the control program to any hardware failures that could affect the successful execution of the control program.

Recovery from machine malfunctions is initially attempted by the hardware instruction retry (HIR) and error checking and correction (ECC) facilities of the hardware. If the hardware recovery attempts are unsuccessful, MCH gets control to analyze the data and isolate the source of error.

When the MCH completes its analysis, it records the error analysis on the SYS1.LOGREC data set and invokes the appropriate functional recovery routines to attempt recovery from the machine check. If recovery is possible, RTM resumes the interrupted program at the point of interruption; if recovery is not possible, RTM terminates the interrupted program.

In a uniprocessing environment, if MCH determines that processing cannot continue, it will terminate execution on the processor and place the processor in a disabled wait state. In a multiprocessing environment, however, an irreparable machine check causes an emergency signal (EMS) to a functioning processor. On this processor, MCH invokes the alternate CPU recovery routine.

Alternate CPU Recovery

The alternate CPU recovery (ACR) routine provides a multiprocessor complex with the ability to recover system operations on an operational processor after another processor fails. Where possible, it will take responsibility for all work in progress on the failing processor.

In a multiprocessing environment, if MCH is unsuccessful because of a recursive error or a damaged processor, MCH invokes ACR on an operative processor to terminate execution on the failing processor. When ACR receives control, it logically removes the failing processor from the system and attempts to transfer work that was in progress on the failing processor to the operative processor. The recovery termination manager then initiates recovery by invoking the appropriate functional recovery routines to free resources associated with the failing processor.

Figure 10-1 demonstrates the flow of control through the machine check handler and alternate CPU recovery.

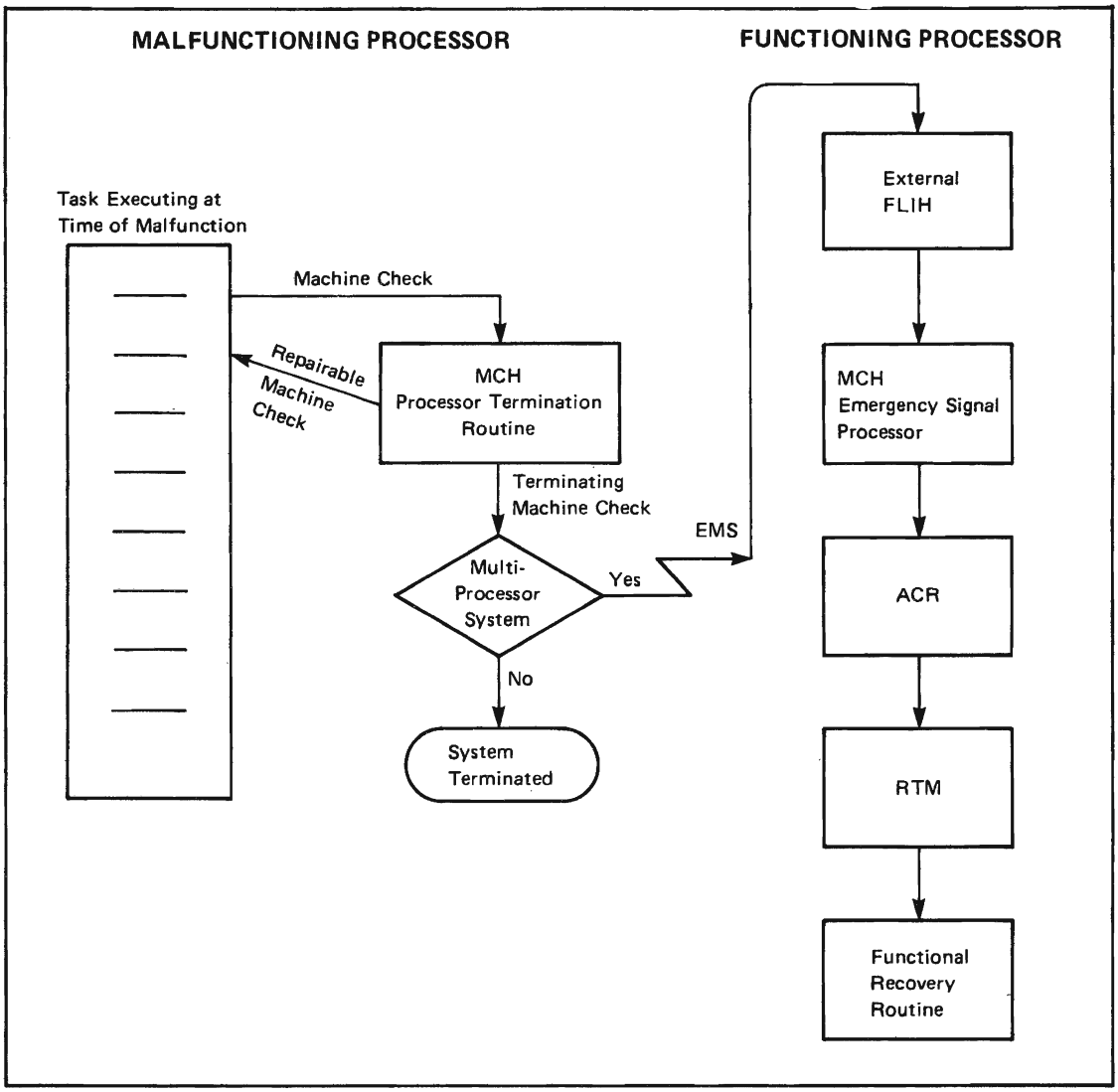


Figure 10-1. Control Flow for MCH and ACR

Subchannel Logout Handler

The subchannel logout handler (SLH) reduces the impact of subchannel malfunctions on systems running MVS/XA. It is an integral part of the I/O supervisor (IOS) that aids in recovering from subchannel errors and informs the operator or system maintenance personnel when errors occur.

SLH receives control after a channel malfunction is detected. It analyzes the type and extent of the error using the information stored by the channel.

When an error condition occurs, SLH allows the device-dependent error recovery procedures to retry the failing I/O, forcing the retry on an alternate subchannel (if one is available). Records describing the error are written to the SYS1.LOGREC data set. SLH performs no error recovery itself; it does not retry any operation or make any changes to the system. Recovery from subchannel errors is performed only by the device-dependent routines.

Dynamic Device Reconfiguration

Dynamic device reconfiguration (DDR) allows the system and user to circumvent an I/O failure by, if possible, moving a demountable volume (tape or disk) from one device to another or by substituting one unit record device (reader, punch, or printer) for another. DDR requests are processed without shutting down the system and might eliminate the need to terminate a job.

Either the system or the operator initiates a DDR swap. When a permanent I/O error occurs, MVS/XA initiates a swap along with a proposed alternate device to take over the processing of the device on which the error occurred. The operator accepts the swap and proposed device, accepts the swap but selects another device, or refuses the swap. The operator can also initiate a swap (via the SWAP command). The ability of the operator to initiate a swap is useful if a device cannot be made ready, if there is a need to substitute one unit record device (such as a card reader or printer) for another, or if, for example, a device must be taken offline for some reason.

Missing Interruption Handler

The missing interruption handler, described in Chapter 7, "Satisfying I/O Requests" also contributes to recovery management in MVS/XA. The missing interruption handler (MIH) checks whether expected I/O interruptions occur within a specified period of time. If an interruption does not occur, MIH notifies the operator so that corrective steps can be taken before system status is harmed. The absence of interruptions might indicate, for example, that there is high contention for a device that may have been reserved for a particular processor, that there is a software problem, or that a device has malfunctioned.

Chapter 11. Initializing the System

Before MVS/XA can do work, it must be initialized. Some starting values are established during the one-time system generation process that occurs when an MVS/XA system is first installed. Others are provided by the system operator during the initialization process that takes place each time the MVS/XA system starts. These values serve to tailor the MVS/XA system to meet the installation's needs.

The system is initialized for several reasons:

- Because of a change in the MVS/XA system
- Because of the installation of a new product
- To resume service after the system has been inoperative

As shown in Figure 11-1 the initialization process consists of loading the nucleus, initializing system resources and resource managers, initializing system component address spaces, and initializing the primary job entry subsystem (usually JES2 or JES3). Input to the process includes data sets initialized during system generation. These data sets reside on the system residence volume (SYSRES) and other direct access device (DASD) volumes. The process also requires the use of real storage, paging data sets, and optional swap data sets. To provide additional system tailoring, the system operator can interact with the various initialization routines through the master console.

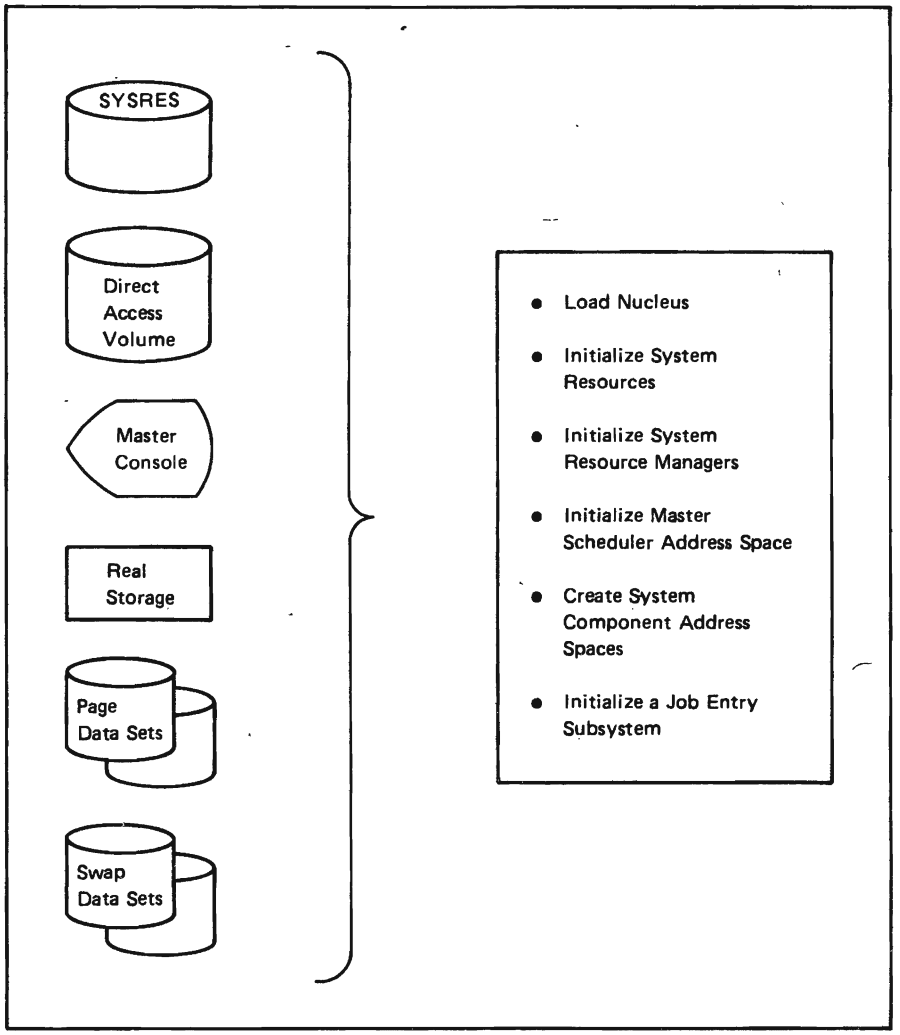


Figure 11-1. The Initialization Process

Loading the Nucleus

System initialization begins when the system operator initializes the hardware. The system operator performs the initial microprogram load (IML) to start the processor, mounts the necessary DASD and tape volumes, and readies the system printers.

The system operator initiates the software load procedures after ensuring that the system residence (SYSRES) volume is mounted. Then, using the master console, the operator activates the LOAD function, which loads the first initialization module into real storage and begins the MVS/XA initialization process.

Initializing System Resources

The system resources initialized include real, virtual, and auxiliary storage, all I/O devices, consoles, and processors. The initialization of system resources is the process of describing what resources are available and in what quantities and setting initial values in system data sets, so that resource managers can control their use. For example, the initialization process provides a description of real storage that the real storage manager (RSM) can use to control the use of real storage.

Initializing the Resource Managers

The initialization process is a **bootstrap operation**. The first initialization control modules must provide for themselves, and for the modules they control, the essential system services that the uninitialized resource managers cannot yet provide. Initialization of a resource manager means initializing the control blocks and the data areas that the manager needs to provide service. When these areas are initialized, the resource manager is ready to perform its tasks.

Initializing the Master Scheduler Address Space

The task of initializing the master scheduler address space continues throughout system initialization. The first step is the mapping of two gigabytes of virtual storage for the master scheduler. This address space will include both **common areas**, which will be available to all system component and user address spaces, and **private areas**, which will be available only to the master scheduler.

Initializing a Job Entry Subsystem

Subsystem initialization is the process of readying a subsystem for use in the system; this process involves defining the subsystem's name and initializing the subsystem so that the system recognizes it by name. In this manner, subsystems such as the job entry subsystem (JES2 or JES3, for example) can be initialized. Subsystems communicate with MVS/XA through a system component known as the **subsystem interface**. Figure 11-2 illustrates how the subsystem interface acts as a liaison between a subsystem and MVS/XA.

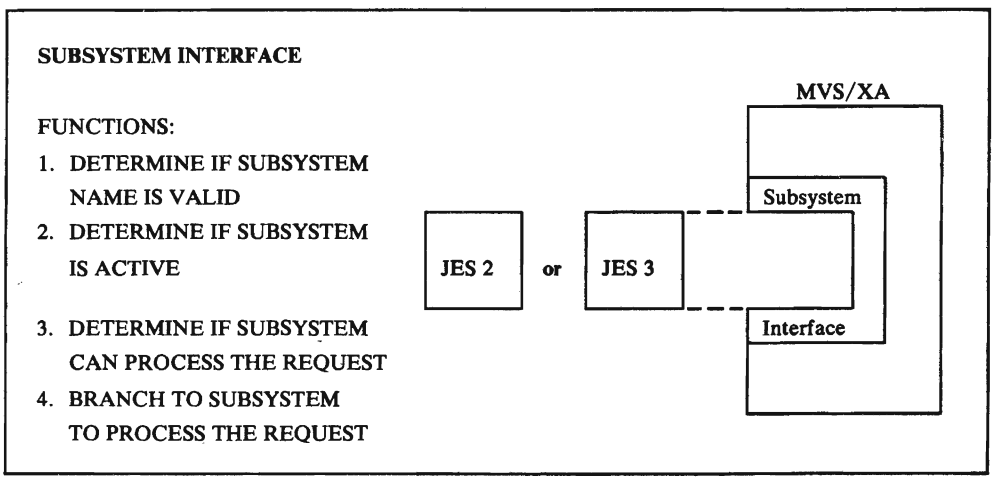


Figure 11-2. The Subsystem Interface

The Initialization Process

The remainder of this chapter describes the sequence of events in the MVS/XA initialization process. It begins with the loading of the **initial program loader (IPL)** control program and ends when the system is ready to accept a LOGON command or a batch job. MVS/XA initialization occurs in three phases. Figure 11-3 summarizes these phases, which are: IPL, NIP, and master scheduler initialization.

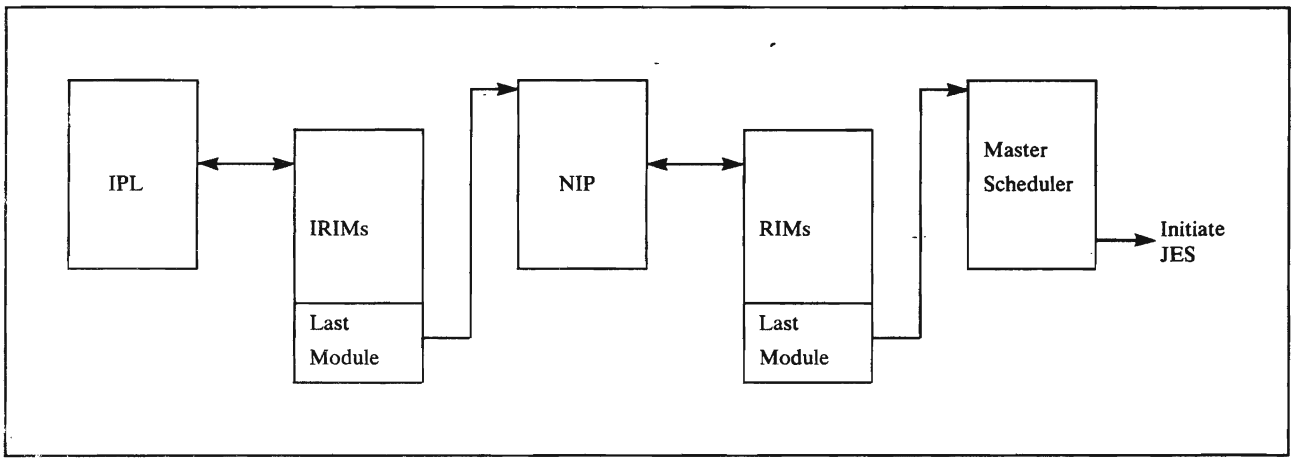


Figure 11-3. System Initialization Summary

1. IPL

   The initial program load (IPL) phase is controlled by the IPL control program, which is loaded when the operator activates the load function. The IPL control program controls the initialization of system resources; however, the actual initialization is done by **IPL resource initialization modules (IRIMs)**. Each IRIM is part of the component that controls a resource. For example, the real storage manager (RSM) IRIM initializes the control blocks that RSM uses and maintains during normal system operation.

2. NIP

   The **nucleus initialization program (NIP)** phase continues the initialization that IPL began. Again, the NIP control program invokes **resource initialization modules (RIMs)** to perform the work of initialization. As with IRIMs, each RIM is part of a component and initializes the control blocks that the component is responsible for.

   During NIP, the PC/AUTH, TRACE, global resource serialization, and DUMPSRV address spaces are initialized.

3. Master Scheduler

   The **master scheduler initialization** routines initialize the master scheduler address space. Once the master scheduler address space is initialized, these routines attach and initialize tasks that remain as permanent system tasks after system initialization.

   During this phase of initialization, the CONSOLE, ALLOCAS, and SMF address spaces are created, and the subsystem interface is initialized. Finally, the master scheduler, itself, starts the job entry subsystem (JES2 or JES3), creating the JES2 or JES3 address space.

Required Resources

During initialization, the control program, the IRIMs, the RIMs, and the master scheduler initialization modules require certain system resources:

- The **system residence volume (SYSRES)** must be online and ready during system initialization because it contains the IPL control program and some of the system data sets needed during the initialization process. Some data sets that must be on the SYSRES volume are: --
  - SYS1.NUCLEUS, which contains the resident nucleus, the IRIMs, the RIMs, the NIP control program, and other initialization modules.
  - SYS1.SVCLIB, which is an authorized program library containing supervisor routines that are not part of the resident nucleus but are invoked during initialization.
- Other **required data sets**, which reside on direct access devices that must be online and ready, are:
  - SYS1.LINKLIB, which contains system and user programs, including the linkage editor, service aid programs, utility programs, and some of the master scheduler initialization modules.
  - The master catalog, which contains pointers to all system data sets.
  - SYS1.PARMLIB, which contains both IBM-supplied and user-supplied lists of system parameter values that serve as input to system initialization. The initialization process depends on values that are specified for the system parameters. System parameters are discussed later in this chapter under, "Processing System Parameters".
  - SYS1.LPALIB, which contains the modules that are loaded into the link pack area.
  - SYS1.STGINDEX, which contains auxiliary storage management (ASM) mapping tables for the pages of virtual input/output (VIO) data sets that must be saved across job steps and between system initializations.
  - SYS1.LOGREC, which is used for recording hardware, software, and I/O errors. This data set is opened during initialization so that error recording can take place.
- The **master console**, which the operator uses to control system initialization. During the NIP phase of initialization, the operator can specify system parameters and override system parameters specified in SYS1.PARMLIB. Because of its importance in operator-system communication, the master console is one of the first devices to be initialized.
- **Real storage**, which must be at least four megabytes. Generally speaking, the more real storage there is, the greater the workload that the system can handle.
- **Page data sets**, which will make up the page space portion of auxiliary storage. The auxiliary storage manager (ASM) uses this page space to store the contents of pageable virtual pages and virtual input/output (VIO) data set

pages. Each page data set is formatted in 4K-byte records called slots. A slot is dynamically allocated whenever a page must be moved out of real storage.

ASM classifies page data sets based on data set content and use. The four types of page data sets are:

- Pageable link pack area (PLPA) data set, which contains system routines and access methods
- Common page data sets, which provide space for the non-PLPA virtual pages in the system common area
- Duplex page data set, which is an optional duplicate data set that an installation supplies as a back-up for the common and PLPA data sets
- Local page data sets, which provide space for each address space's unique pages, the virtual input/output (VIO) data sets, and, if there are no swap data sets available, private address space (LSQA) pages

- **Swap data sets**, which make up the swap space portion of auxiliary storage. ASM uses swap space to store and retrieve the set of pages that belong to a swapped-out address space.

  Swap data sets are optional. However, if none are supplied, swapping is done to page data sets which, can degrade system performance.

  Each page and swap data set must be defined and cataloged on a direct access storage device (DASD). While the system is running, these data sets must remain open.

Initial Program Load (IPL)

IPL is the first phase of the system initialization process. When the operator initiates the load process, a bootstrap loader brings the IPL control program into real storage starting at location zero, as shown in Figure 11-4. Then the IPL control program receives control.

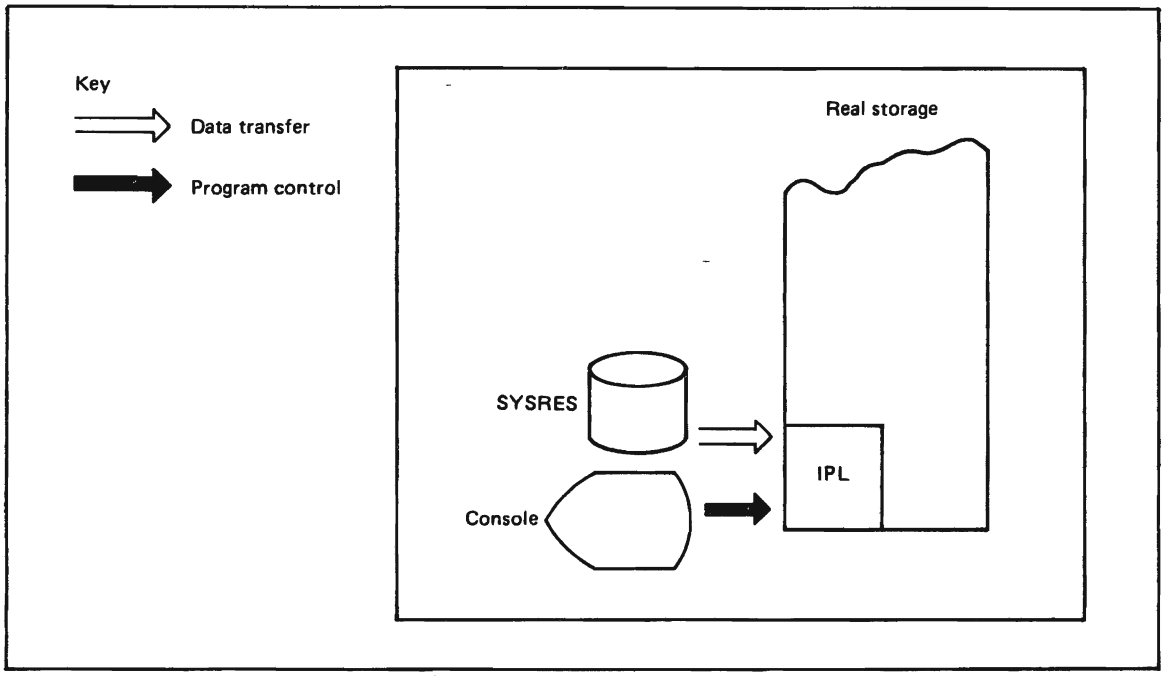


Figure 11-4. Loading the IPL Control Program

The IPL Program

The IPL program prepares an environment in which the IRIMs can execute, controls the loading and deleting of the IRIMs, and provides service routines for the first phase of initialization. It handles page faults by assigning frames of real storage to each page of virtual storage that the IRIMs request. It clears real storage, and maps two gigabytes of virtual storage for the master scheduler address space. It searches SYSRES for the SYS1.NUCLEUS data set, which contains the IRIMs, RIMs, and other modules needed for system initialization. The IPL control program then passes control to the first IRIM.

The Work of the IRIMs

As stated earlier, the IRIMs are the programs that actually do the work of the IPL phase. They perform very basic initialization tasks; they load the nucleus, build virtual and real storage areas, and initialize the device on which SYSRES resides.

Loading the Nucleus

As Chapter 2, "Multiple Virtual Storage" explains, some code in the nucleus must run with dynamic address translation (DAT) turned off. Because of this requirement, the nucleus consists of two load modules, which the first IRIM loads into storage. The IRIM places the DAT-off nucleus in contiguous frames of the highest available addresses of real storage; it is not mapped in virtual storage. The IRIM loads the control sections (CSECTs) in the four sections of the DAT-on nucleus into virtual storage from low addresses upward in the order of: read-write, read-only, read-only extended, and read-write extended.

An IRIM builds the DAT-off to DAT-on linkage table at the beginning of the DAT-off nucleus in real storage. This table establishes addressability between entries in the DAT-off nucleus in real storage and entries in the DAT-on nucleus in virtual storage.

One of the IRIMs builds the nucleus map (NUCMAP), an address-sorted directory of CSECTs and entry points in the DAT-on nucleus. NUCMAP resides in virtual storage in the read-write extended nucleus.

Initializing Virtual Storage

During the IPL phase of initialization, the IRIMs initialize or reserve storage for many system component control blocks, work areas, and programs. The IRIMs also begin to initialize the private area of the master scheduler address space, the first address space to be created. Some important areas initialized in this first address space are:

- The system queue area (SQA) and extended SQA

  The VSM IRIM reserves storage for the tables and queues that relate to the system.

- The extended local system queue area (extended LSQA) for the master scheduler

  The VSM IRIM initializes the area above the 16-megabyte line that contains tables and queues that the master scheduler will use.

- The master scheduler segment table

  An RSM IRIM initializes a segment table whose entries are the addresses of page tables for the common area of virtual storage. This common segment table is part of the master scheduler address space segment table. During the NIP phase of initialization, an RSM RIM copies the common segment table from the master scheduler's private area into SQA for all address spaces to use. Eventually, each address space also has a segment table for its own private area.

- The RSM page frame table

  An RSM IRIM initializes the tables that identify how the frames of real storage are assigned. As Chapter 2, "Multiple Virtual Storage" explains, there is one page frame table for the entire system, and it has one entry for each frame of real storage. This table resides in the read-write extended nucleus. The IRIM initializes those frames that have already been assigned and are permanently resident.

Figure 11-5 shows the virtual storage map for the master scheduler address space at the end of the IPL phase.

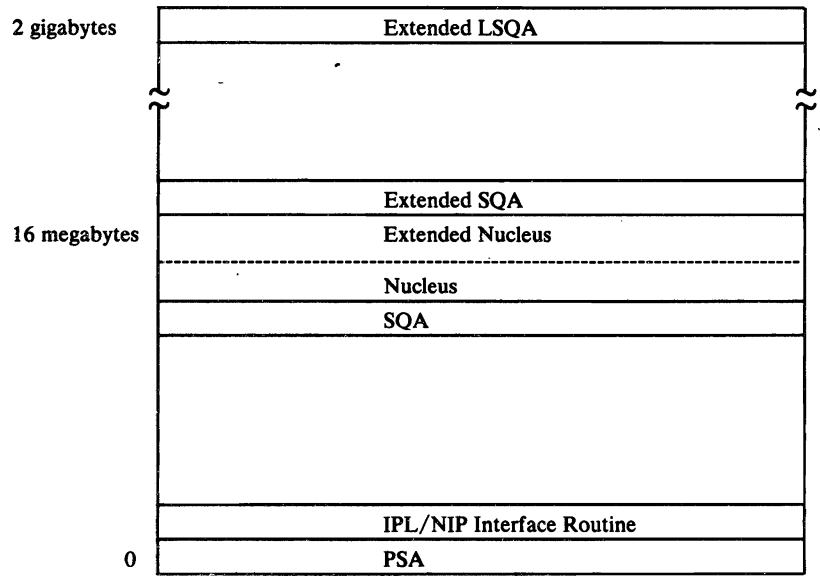


Figure 11-5. Virtual Storage at Exit from the IPL Phase of Initialization

Initializing Real Storage

As previously described, the IPL control program zeroes each 4K-byte frame of real storage. It then reserves space for permanent data areas and control blocks. By the end of the IPL phase, the following areas have been permanently initialized in real storage:

- System data areas that are never paged out of real storage
- DAT-off nucleus
- DAT-on nucleus
- Prefix save area (PSA)

Figure 11-6 shows a schematic representation of real storage at the end of the IPL phase of initialization. The figure illustrates the following points.

- As the virtual storage manager (VSM) initializes the DAT-on nucleus, the SQA, the extended SQA, and the extended LSQA in virtual storage, RSM allocates non-contiguous real storage frames to provide working copies of these pages.
- The PSA resides at location zero.
- Only the DAT-off nucleus resides in contiguous real frames.
- The code for the last executing module (the last IRIM) is in real storage at exit from IPL.

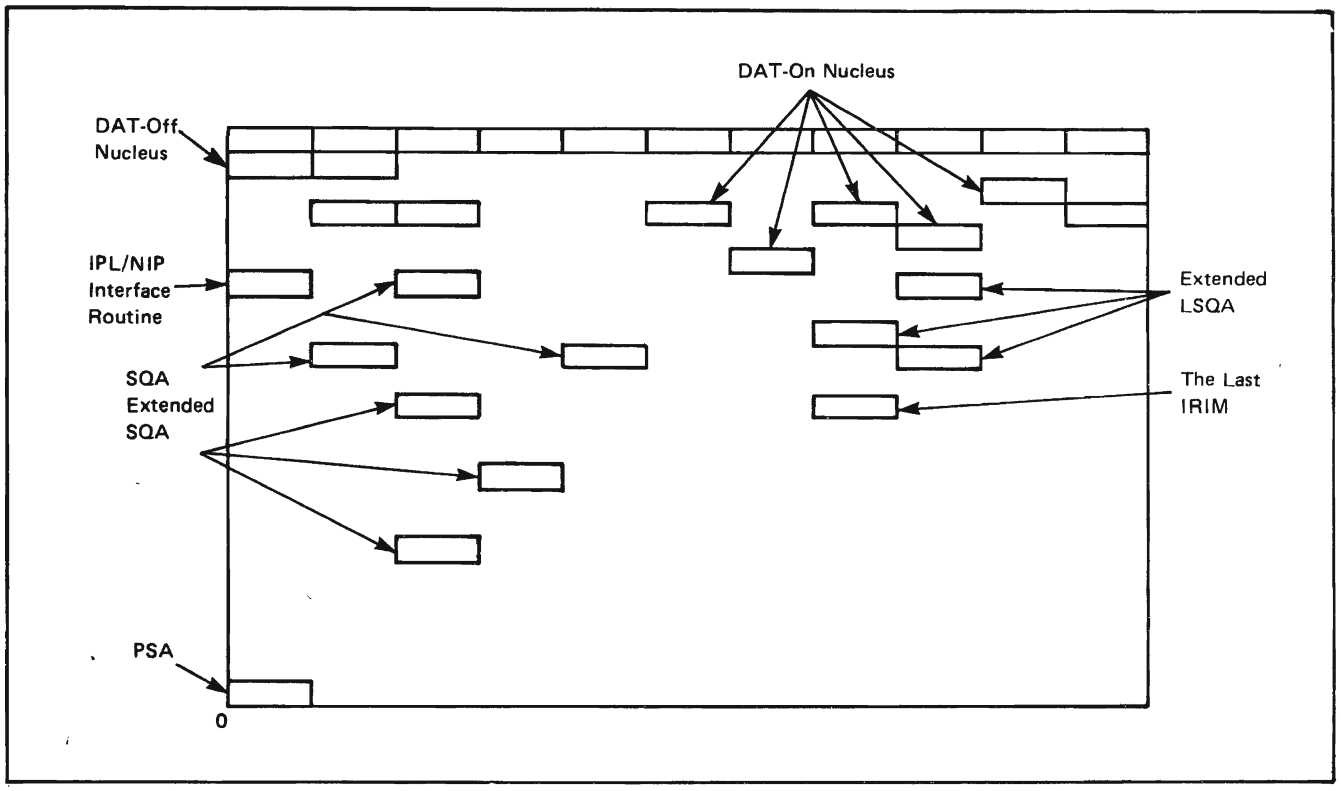


Figure 11-6. Real Storage at Exit from IPL

Initializing the IPL Device

One of the IRIMs initializes the unit control block (UCB) for the device on which the IPL volume, SYSRES, resides. Note that the IPL device is only the first of many devices to be initialized.

Nucleus Initialization Program (NIP)

NIP processing is the second phase of the system initialization process. The NIP control program prepares the environment that will allow the resource initialization modules, the RIMs, to perform their functions of initializing system components. The NIP control program is responsible for the loading and deleting of the RIMs. It also provides service routines that substitute for functions that are not yet available in the system, as well as diagnostic support for software and hardware failures that might occur during this second phase.

The RIMs depend on system parameters to tell them what initialization functions to perform and which SYS1.PARMLIB members to use to initialize the system. System parameter lists are contained in the IEASYSxx member of SYS1.PARMLIB or are specified by the operator during NIP. Based on the values in effect for the system parameters, the RIMs perform three major functions:

- Continue to establish the master scheduler address space
- Process SYS1.PARMLIB-specified and operator-specified system initialization parameters
- Continue to initialize the resource managers

Establishing the Master Scheduler Address Space

NIP and the RIMs establish the master scheduler address space, completing the job started during the IPL phase. As stated earlier, the master scheduler address space contains private areas for the master scheduler (in which NIP and the RIMs execute) and common areas for use by all address spaces. As shown in Figure 11-7, VSM and ASM RIMs allocate virtual storage in the common area for the common service area (CSA), the system queue area (SQA), and the link pack area (LPA). All of these areas exist both below the 16-megabyte line (for programs executing in 24-bit addressing mode) and above the 16-megabyte line (for programs executing in 31-bit addressing mode). Figure 11-7 shows the completed map of the master scheduler address space.

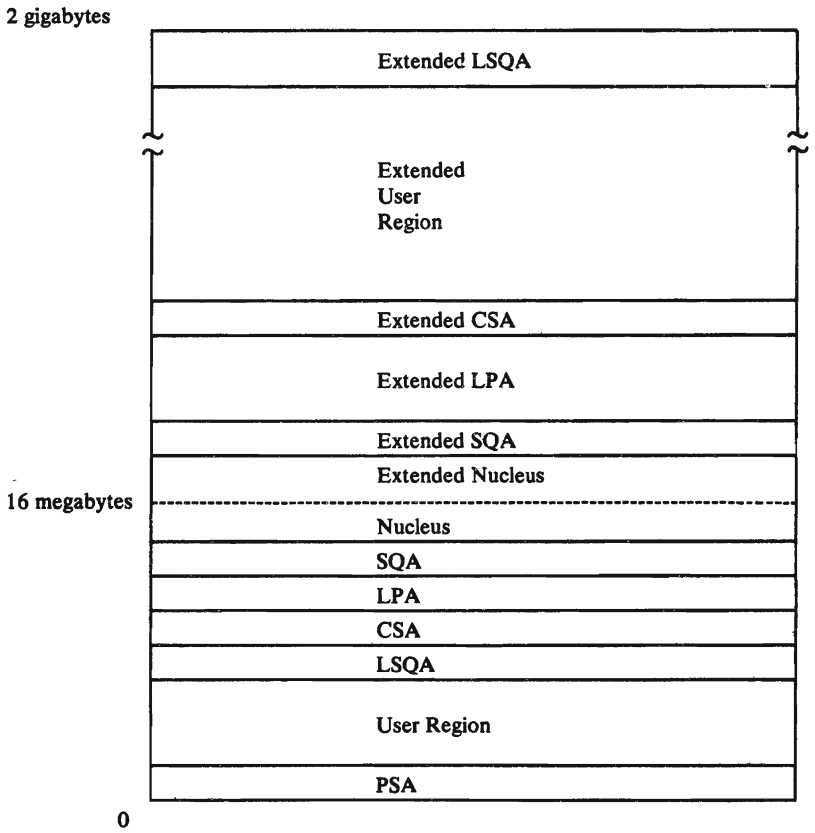


Figure 11-7. Virtual Storage at Exit from NIP

The amount of storage allocated for the different areas of virtual storage depends on values specified for system parameters. An example of how the VSM RIM uses a system parameter appears in the following section on processing system parameters.

Processing System Parameters

During NIP processing, the RIMs depend on system parameters to tell them how to perform certain initialization functions. The RIMs obtain system parameters from two sources: from system parameter lists, which reside in SYS1.PARMLIB, and directly from the system operator.

System Parameter Lists

System parameter lists reside in SYS1.PARMLIB. The NIP RIMs always read the **default general parameter list** (IEASYS00). This list contains basic initialization instructions, installation-specified initialization defaults, and other initialization values that will not change from one initialization to another. For example, if IEASYS00 contains

```
CSA=(400,2000)
```

a VSM RIM allocates 400K bytes for the CSA and 2000K bytes for the extended CSA.

SYS1.PARMLIB may also contain **alternate system parameter lists** (IEASYSxx members other than IEASYS00) that NIP merges with the default parameter list during initialization. The alternate parameter lists, sometimes called secondary lists, contain values that override corresponding values in the default list. They may also contain additional values not originally specified in the default list. Alternate lists normally contain parameters that are subject to change. For example, they might contain parameters that, because of workload changes, must change between shifts.

System Operator

After console communication has been established, the system operator receives the message:

```
SPECIFY SYSTEM PARAMETERS
```

If an installation wants NIP to merge one or more alternate parameter lists with the default list, the system operator identifies them at this time. In addition, the system operator may directly specify certain system parameters. Such a "direct specification" would include parameters that are unique for a specific initialization. If no alternate parameter lists or direct specifications are indicated by the system operator, the default general system parameter list is the sole source of initialization values.

The operator selects the type of initialization, which affects the data sets that are opened and their starting values. The type selected depends upon the reason the system has previously been shutdown:

- The **cold start**, the most complicated initialization process, involves reloading the link pack area, respecifying page and swap data sets, and deleting previous VIO data set pages. It is performed (by specifying the CLPA system parameter) under these circumstances:
  - If this is the first initialization after system generation
  - If the installation is adding or modifying modules in SYS1.LPALIB
  - If the link pack area data set pages need to be restored
- The **quick start**, the usual initialization performed after normal shut-down, uses the link pack area from the previous system initialization without reloading it. The VIO data set pages are deleted, page data sets can be added, and swap data sets must be respecified.
- The **warm start**, the simplest initialization process, occurs after a system has completed a cold start but then has a system failure. The VIO data set pages

are retained, page data sets can be added, and swap data sets must be respecified.

Initializing System Resources and Resource Managers

Many resources and resource managers are initialized by the RIMs in the second phase of system initialization — in fact, too many to describe here. To give examples of the type of processing the RIMs perform, this chapter will describe the initialization of:

- I/O devices
- System catalog
- Auxiliary storage management (ASM)

Initializing I/O Devices

To initialize the devices in the configuration the input/output supervisor (IOS) RIMs need to initialize two data areas: the unit control blocks (UCBs) and the installed channel path table (ICHPT).

Each device is represented by a UCB that MVS/XA uses for device allocation and for controlling input/output operations. The IOS RIMs initialize the UCB for each device by setting status and condition flags. For direct access devices (DASDs), the IOS RIM also records volume information in the UCBs. Initialization of the UCBs requires several steps:

- Initializing the channel subsystem

  Every device has one subchannel in every system to which it is attached. An IOS RIM initializes all valid subchannels by placing the subchannel number in the UCB of the corresponding device and enabling the appropriate subchannel.

  Once the channel subsystem is initialized, an IOS RIM initializes the installed channel path table (ICHPT) to reflect the current state of each channel path, such as whether or not the channel path is online.

- Testing the availability of a device

  The IOS RIM considers a device unavailable if it was (1) defined as offline during system generation or (2) defined as online but does not now have an available channel path.

- Testing the accessibility of a device

  The IOS RIM tests the accessibility of each available device on all defined channel paths. Figure 11-8 illustrates a configuration in which I/O device X has a single channel path, and devices Y and Z have multiple channel paths. For a device to be accessible, there must be at least one channel path to that device.

  To test for accessibility, an IOS RIM requests an I/O operation on each defined channel path. The results of these I/O operations determine how a device can be accessed. For a DASD, the IOS RIM first verifies that there is an available channel path by issuing a dummy (no-op) command to determine whether it can communicate with the device. If so, the IOS RIM reads the volume label to determine the volume serial number and the location of the

volume table of contents (VTOC). For a shared DASD, an IOS RIM requests an I/O operation to determine if the device is actually shareable. Unavailable devices are not tested for accessibility.

- Checking for duplicate volumes

  As the DASD UCBs are initialized, an IOS RIM scans the UCBs for online DASDs to verify that there are no duplicate volume serial numbers. If any duplicate volumes are found, a message asks the operator to remove one of them.

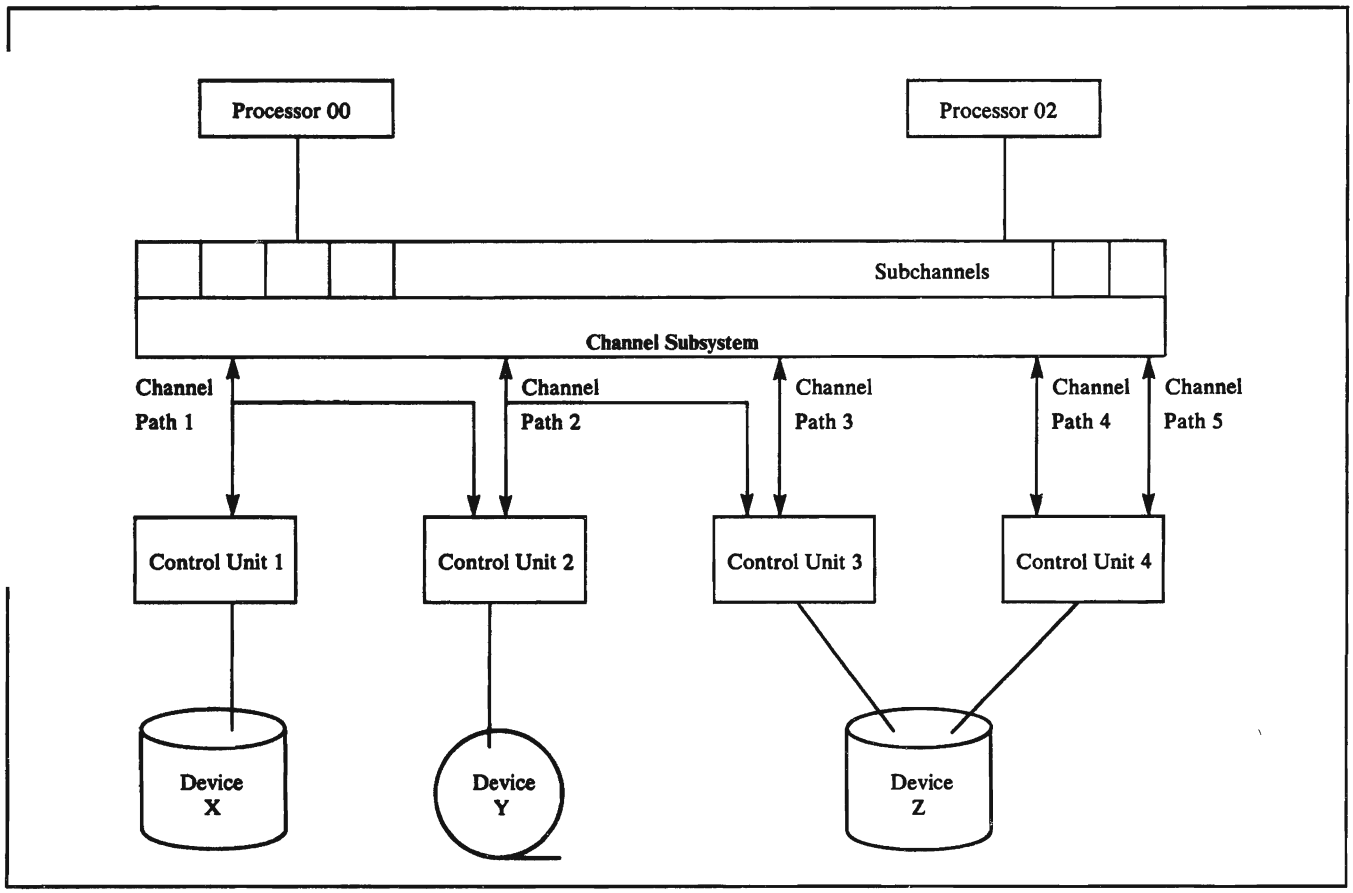


Figure 11-8. Initializing Channel Paths

Initializing the Master Catalog

The master catalog is used to locate cataloged data sets and other catalogs. An entry for a cataloged data set contains the volume serial number and device type. The master catalog can contain entries for VSAM and non-VSAM data sets and VSAM and integrated catalog facility (ICF) user catalogs.

During NIP, initialization routines can open data sets residing on the system residence volume whether or not the master catalog has been opened. However, they must use master catalog pointers to locate system data sets residing on volumes other than the system residence volume; and they cannot open or access these data sets until the master catalog is initialized. For example, before NIP can complete the opening of SYS1.LINKLIB and read from SYS1.PARMLIB, NIP must open the master catalog.

RIMs open, initialize, and close the master catalog at initialization time. The system operator receives the message:

```
SPECIFY MASTER CATALOG PARAMETER
```

and must identify the SYS1.NUCLEUS member that contains the volume serial number and the device type of the desired master catalog. As shown in Figure 11-9 the RIM then locates the UCB representing the device on which the volume is mounted. If the volume containing the master catalog is not mounted, the RIM issues a message that asks the operator to mount it. A RIM builds the necessary control blocks, then opens the data set and initializes it as the master catalog.

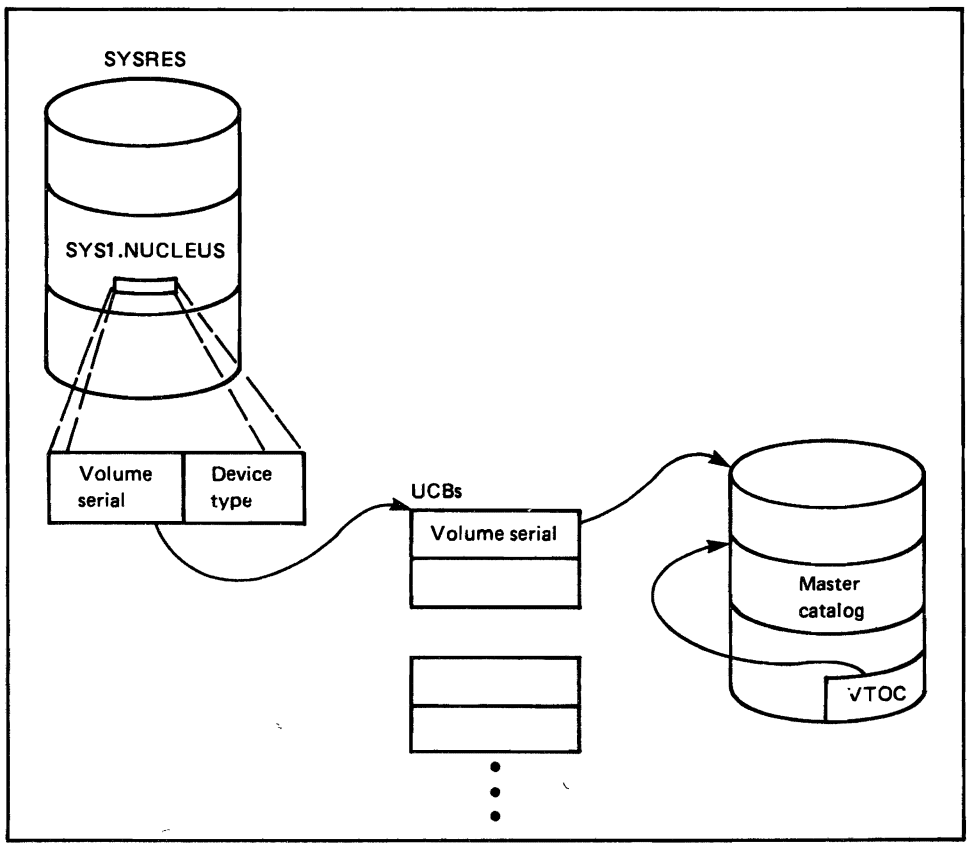


Figure 11-9. Locating the Master Catalog

After NIP processing finishes but before NIP terminates, it invokes a RIM to close the master catalog. After system initialization is complete, the first reference to a cataloged system data set causes the master catalog to be opened for normal use.

Initializing the Auxiliary Storage Manager

The auxiliary storage manager (ASM) controls the auxiliary storage the system uses for paging and swapping and requests I/O operations needed for paging and swapping. To page efficiently, ASM divides paging requirements into pageable link pack area (PLPA), common, local pages and duplex. During system generation, the installation allocates, catalogs, and formats page data sets to meet its requirements for the four types of page data sets. System generation places the names of the data sets into the default system parameter list. Additional page data

sets can be specified in the alternate system parameter lists or supplied directly by the system operator at system initialization.

Optionally, the names of installation-defined swap data sets and/or duplex (duplicate) data sets can be specified in the same manner. Also, the installation can indicate whether it wants VIO data sets to be reestablished when a subsequent system initialization is performed.

After initialization, additional page and swap data sets can be dynamically added to the system. To do this, the system operator uses the PAGEADD command and names the the page or swap data sets to be added.

Initializing Page Data Sets

The ASM RIM opens and initializes page data sets according to the type of IPL start — cold, quick, or warm (described earlier in this chapter under "System Operator"). During a cold start, the PAGE system parameter in the default system parameter list specifies applicable page data set names. However, during IPL, the operator also receives the message:

```
SPECIFY SYSTEM PARAMETERS
```

and can use the PAGE parameter to specify additional page data sets.

During a quick start, the link pack area is rebuilt, not reloaded. That is, the page and segment tables are reset to match the last-created link pack area. The PAGE system parameter in the default system parameter list, or the operator response to system messages, supplies the applicable page data set names.

During a warm start, the page data set names are those used in the previous system initialization, although the operator can use the PAGE parameter to specify additional data sets.

Successful initialization of ASM requires that one PLPA, one common, and at least one local page data set, be specified and available. Before initialization, however, all page data sets (up to a maximum of 64) must be allocated, cataloged, and formatted as VSAM data sets.

The installation can, optionally, define a duplex data set that holds a duplicate copy of all pages written to the pageable link pack area (PLPA and extended PLPA) and common page data sets. The DUPLEX system parameter, contained in a system parameter list or specified directly by the system operator, specifies the duplex data set name.

Initializing Swap Data Sets

Swap data sets are optional, but their use can significantly improve performance. (If no swap data sets are specified, and swapping occurs, ASM directs the LSQA and working set pages of the swapped-out address space to a local page data set.). Swap data set names are specified by the SWAP system parameter in one of the system parameter lists or supplied directly by the operator. Unlike the PAGE parameter, the SWAP parameter is an overriding parameter that permits the replacement of data set names specified in the system parameter lists.

Initializing the Master Scheduler

Master scheduler initialization is the third and final phase of the system initialization process. As shown in Figure 11-10, it consists of three steps:

1. Initializing the master scheduler base
2. Initiating the master scheduler
3. Initializing the master scheduler region

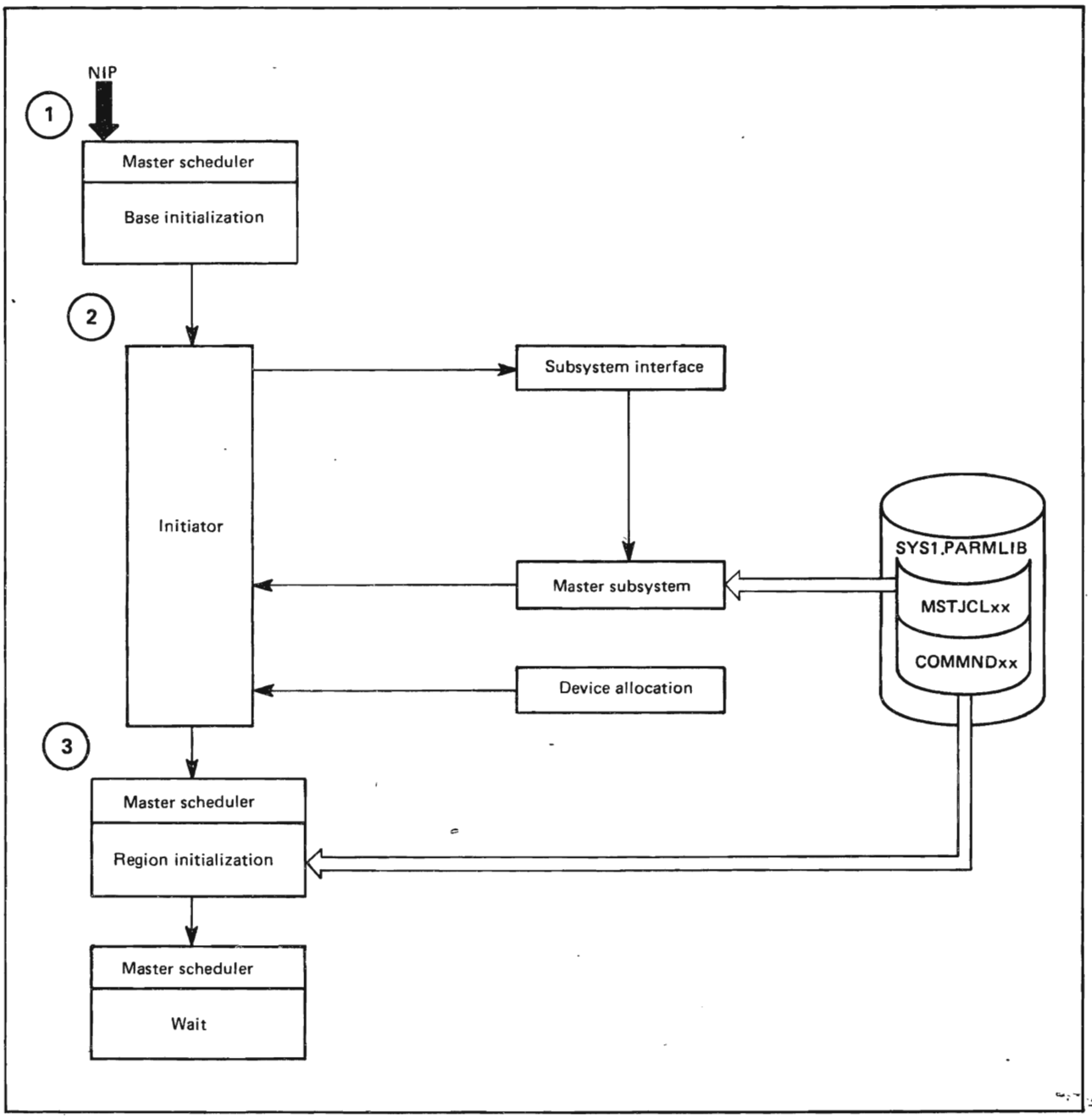


Figure 11-10. Master Scheduler Initialization

Initializing the Master Scheduler Base

The master scheduler base initialization routine is entered from NIP processing. It loads routines required by system-initiated cancel, SWA management, and resource management routines. It creates and initializes the control blocks needed to invoke the initiator, a system program that starts a job step. Then, it locates and stores entry points for certain job scheduler routines. It initializes the subsystem interface, the communications task, and some TSO addresses and parameters. It performs master trace initialization, sets the time-of-day clocks, and, finally, it attaches the initiator to initiate the master scheduler.

Initiating the Master Scheduler

Before attaching the master scheduler region initialization routines that initiate the master scheduler, master scheduler initialization starts tasks that remain as a permanent system tasks. These include the missing interruption handler (MIH) and error recovery routines. At this point, no JES readers are active and no procedure libraries are open. So, the initiator gets the JCL necessary for attaching the master scheduler region initialization routines from the MSTJCLxx member of SYS1.PARMLIB.

To read and process MSTJCLxx, the initiator invokes, through the subsystem interface, the **master subsystem** - a primitive job entry subsystem. The master subsystem, reads MSTJCLxx and invokes job scheduler routines to process the JCL and initialize the necessary control blocks. The last statement in MSTJCLxx is a command to START JES. This command is passed to the command processor portion of the master scheduler and scheduled for execution.

The initiator uses the device allocation routines to allocate the data sets indicated in MSTJCLxx and required by the master scheduler (data sets such as SYS1.PROCLIB and SYS1.PARMLIB). These data sets are required when JES is subsequently started. Two internal reader data sets are also allocated. They are used later to pass JCL from system routines to JES. Lastly, the initiator attaches master scheduler region initialization as the job step task. The master scheduler is now active.

Initializing the Master Scheduler Region

The region initialization routine attaches other tasks to be run in the master scheduler region and passes commands found in SYS1.PARMLIB to the command processor for execution or scheduling. These commands are contained in a command list (COMMNDxx, a member of SYS1.PARMLIB). Because there can be multiple command lists, the CMD system parameter is used to tell master scheduler initialization which list to use.

After master scheduler initialization completes, control passes to the master scheduler; the master scheduler waits for individual system commands to be issued and then activates the processing of each command. When the START JES command appears in MSTJCLxx, the master scheduler starts the initialization of the job entry subsystem.

Initializing the Job Entry Subsystem

The process of initializing the job entry subsystem (JES) consists of:

1. Creating an address space for JES
2. Initializing a region control task (RCT) to ready the JES address space for execution
3. Building JCL statements that invoke the JES initiation procedure
4. Passing the JCL to an initiator

Creating an Address Space for JES

Creating an address space for JES is similar to creating any address space. The master scheduler attaches the address space create routine. This routine asks the system resources manager (SRM) if a new address space can be created, and, upon receiving permission to proceed, builds LSQA in the private area and initializes segment tables and page tables to represent the new address space. Lastly, the address space create routine builds task control blocks for a region control task (RCT) and places the address space control block (ASCB) on the dispatching queue. When JES3 is the primary job entry subsystem, a second JES3 address space, JES3AUX, can be created after master scheduler initialization completes.

Initializing the Region Control Task

The region control task (RCT) is the highest priority task in the new address space. RCT controls the address space and prepares it for execution. It has responsibility for attaching the started task control (STC) routine and managing the swapping activity of the address space. Therefore, when the JES address space becomes active, the first task dispatched is the RCT. After the RCT is initialized, it attaches the STC to initiate JES.

Initiating JES

The START JES command causes the STC routine to build the JCL necessary to invoke the JES procedure. Then STC starts the job entry subsystem.

The initiator invokes the master subsystem, which uses job scheduler routines much as it did when initiating the master scheduler. However, to start JES, the initiator uses the internal JCL built by STC rather than MSTJCLxx.

After all JCL has been processed and after job scheduler control blocks have been built in the SWA, the initiator calls device allocation to allocate the data sets specified in the JES procedure. Then, using the program name from the EXEC statement of the JES procedure, the initiator attaches the primary job entry subsystem. JES is started and MVS/XA is ready for work.

Initializing the Time Sharing Option (TSO)

When TSO sessions are part of an installation's workload, TSO must be initialized before TSO logons can be accepted. TSO initialization requires two steps.

First, an operator START command starts the telecommunication access method (TCAM or VTAM) selected by the installation. (These telecommunication access methods are described in more detail in Chapter 7, "Satisfying I/O Requests") The

master scheduler recognizes the START command and creates an address space for the access method.

For TCAM, the operator must next enter the MODIFY command to activate the terminal I/O controller (TIOC) as a subtask of TCAM.

For VTAM, the operator must enter a second START command to activate the terminal control address space (TCAS). Once either of these commands has been processed, TSO users can LOGON.

Creating User Address Spaces

Batch jobs, entered by means of card readers or similar unit record devices, the TSO SUBMIT command, or other running jobs, run in an initiator's address space. Jobs entered by means of the START, MOUNT or LOGON command run in their own address space.

The operator issues a START command to run any of the user or IBM-supplied programs whose JCL is stored in the system library, SYS1.PROCLIB. The operator issues the MOUNT command to run programs that affect the attributes of I/O devices such as whether they are available for public or private access. All system users with a TSO identification give the LOGON command to begin an interactive computing session at a terminal.

When a START, MOUNT, or LOGON command is issued, the master scheduler uses other system components to create a new address space and a task that performs the requested function in the address space. Figure 11-11 summarizes the process of creating an address space.

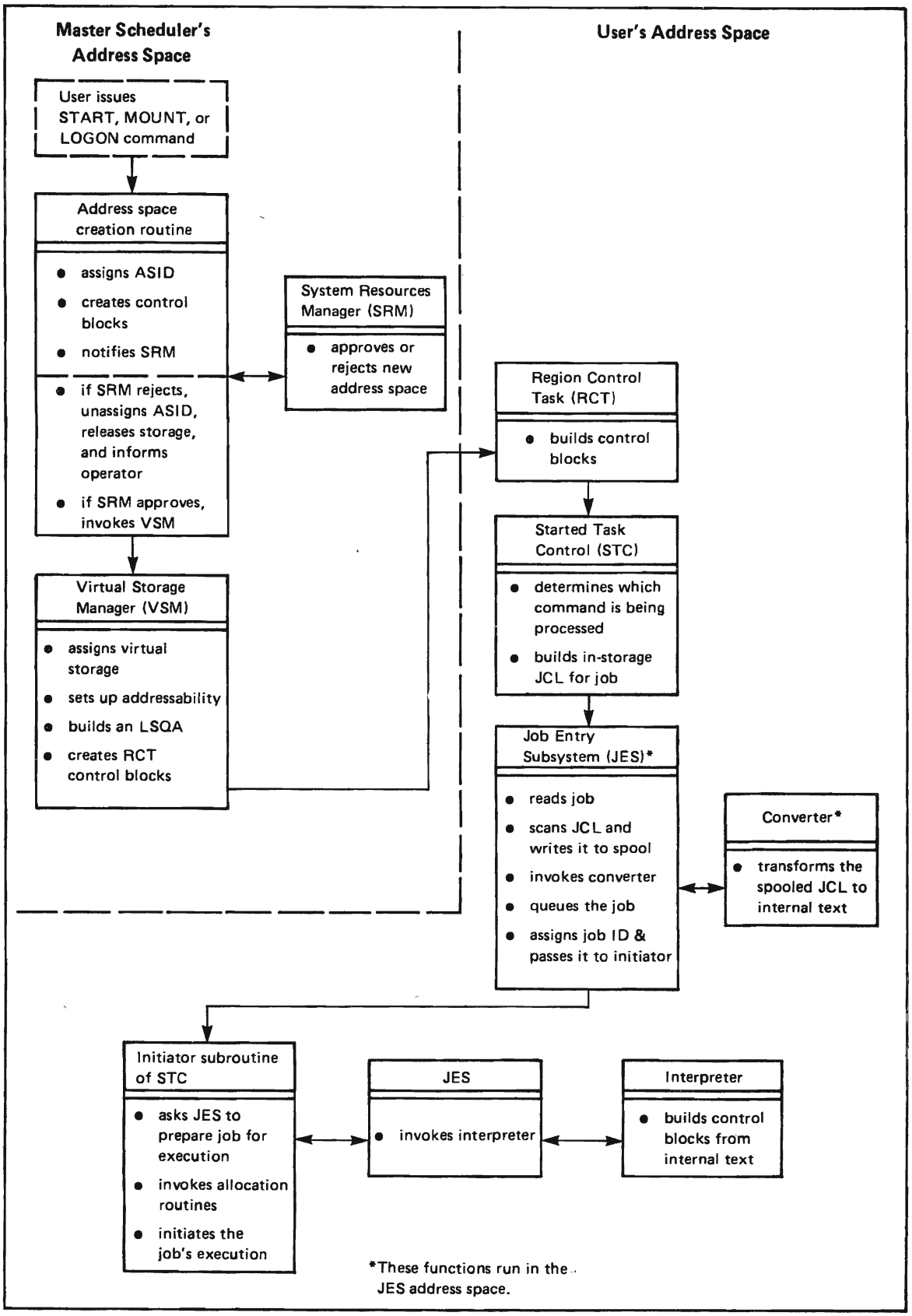


Figure 11-11. Creating an Address Space

The **address space creation** routine, operating in the master scheduler's address space, assigns an address space identifier (ASID) to the new address space and creates control blocks for it. Then the routine notifies the system resources manager (SRM) that a new address space is to be created. SRM decides (based on the availability of system resources) whether the creation of an address space should be allowed.

If system conditions are unfavorable for creating a new address space (such as when there is a shortage of auxiliary storage, pageable frames, or SQA), SRM does not allow the address space to be created. Instead, the address space creation routine releases the ASID and frees the storage used by the control blocks. The operator receives a message indicating that the address space could not be created.

If current system conditions are favorable to creating the new address space, the address space creation routine invokes the virtual storage manager (VSM) to assign virtual storage and set up addressability for the address space. VSM builds a local system queue area (LSQA) and calls RSM to set up a segment table, a page table, and external page tables in it. VSM also creates control blocks to operate the region control task (RCT) for the address space.

Next, the RCT receives control in the new address space. There is one RCT in each address space. When the address space is created, the RCT is the only task associated with it. The RCT builds control blocks that further define the address space, then attaches the started task control (STC) routine.

STC determines which command is being processed (START, MOUNT, or LOGON), builds in-storage JCL for the task associated with the command, then passes the JCL to the job entry subsystem.

For a TSO user, the LOGON initialization routine verifies all the user-supplied LOGON parameters, prompts the user for any additional ones, and builds the JCL necessary to invoke the LOGON procedure. LOGON initialization then passes this JCL to the job entry subsystem.

The job entry subsystem reads the job, scans the JCL and writes it on a spool data set, invokes the converter to transform the spooled JCL into internal text, queues the job on an internal queue, and assigns a job ID, which it returns to STC.

Next, STC uses its **initiator subroutine** to pass this job ID back to the job entry subsystem with a request to prepare the job for execution. The job entry subsystem invokes the interpreter to build and initialize the scheduler control blocks for the address space from the internal text created by the converter. Upon return from the job entry subsystem, the initiator subroutine invokes the allocation routines and then issues an ATTACH macro instruction for the task related to the address space: any STARTed program, the MOUNT command processor (MOUNT), or the **terminal monitor program (TMP)** for a LOGON request.

TMP is the program that controls the interchange of user commands with TSO. After the TMP starts, and you, the TSO user, have logged on, the READY message appears; MVS/XA awaits your command.

Index

MVS/Extended Architecture
Overview

GC28-1348-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: ***Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.***

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

_____

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

Cut or Fold Along Line

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

GC28-1348-0

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT 40 ARMONK, NEW YORK

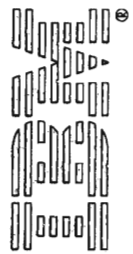
POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department D58, Building 920-2
PO Box 390
Poughkeepsie, New York 12602

Fold and tape | Please Do Not Staple | Fold and tape

MVS/Extended Architecture Overview (File No. S370-34) Printed in U.S.A. GC28-1348-0

MVS/Extended Architecture Overview (File No. S370-34) Printed in U.S.A. GC28-1348-0



GC28-1348-0