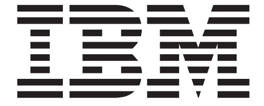
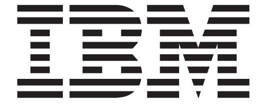
z/OS Basic Skills Information Center



z/OS concepts

z/OS Basic Skills Information Center



z/OS concepts

Note

Before using this information and the product it supports, read the information in “Notices” on page 107.

This edition applies to z/OS (product number 5694-A01).

We appreciate your comments about this publication. Comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Send your comments through this Web site: http://publib.boulder.ibm.com/infocenter/zoslnctr/v1r7/index.jsp?topic=/com.ibm.zcontact.doc/webqs.html

© Copyright International Business Machines Corporation 2006, 2008.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction to z/OS v

Chapter 1. z/OS operating system: Providing virtual environments since the 1960s 1
Hardware resources used by z/OS 1
Multiprogramming and multiprocessing 3
z/OS programming constructs: Modules, macros, components and control blocks 3
Physical storage used by z/OS 4
What is virtual storage? 5
What is an address space? 6
What is dynamic address translation? 7
How z/OS uses physical and virtual storage . . 7
The role of storage managers 12
A brief history of virtual storage and 64-bit addressability. 13
What is meant by “below-the-line” storage? . . 17
What’s in an address space? 17
System address spaces and the master scheduler 19
What is workload management? 20
I/O and data management 22
Supervising the execution of work in the system . . 22
What is interrupt processing? 23
Dispatchable units of work: Tasks and service requests 25
Preemptable versus non-preemptable units of work 26
What does the dispatcher do? 27
Serializing the use of resources 28
Defining characteristics of z/OS 30
Additional software products for z/OS 31
Middleware for z/OS 32
A brief comparison of z/OS and UNIX 32

Chapter 2. z/OS storage constructs: File systems, data sets, and more . . . 35
What is a data set? 36
Quick reference: Data set structure. 37
Where are data sets stored? 38
What are access methods? 38
What are DASD volumes and labels?. 39
Allocating a data set 40
Types of data sets 44
What happens when a data set runs out of space? 47
What is VSAM? 48
What is a VTOC? 49
What is a catalog? 50
What is a generation data group? 53
Role of DFSMS in managing space 53
z/OS UNIX file systems 55
z/OS data sets versus file system files 56
What is a zFS file system? 57

Chapter 3. Interacting with z/OS: TSO, ISPF, and z/OS UNIX interfaces 59
What is TSO?. 59
What is TSO native mode? 60
How are CLISTs and REXX used? 62
What is ISPF? 62
ISPF keyboard keys and functions. 66
The ISPF Data Set List utility 68
The ISPF editor 69
The ISPF Settings menu 71
What is z/OS UNIX? 72
ISHELL command (ish) 74
OMVS command shell session 75
Direct login to the z/OS UNIX shell 76

Chapter 4. Processing work on z/OS: How the system starts and manages batch jobs 79
What is batch processing? 79
What is JES? 80
What does an initiator do? 82
Batch processing and JES: Scenario 1 83
Batch processing and JES: Scenario 2 88
Job flow through the system. 89
JES2 compared to JES3. 91

Chapter 5. Doing work on z/OS: How you submit, control and monitor jobs using JCL and SDSF. 93
What is JCL? 93
How is a job submitted for batch processing? . . . 95
What is the System Display and Search Facility (SDSF)?. 95

Chapter 6. Parallel Sysplex: Worth the effort for continuous availability . . . 101
Benefits of Parallel Sysplex: No single points of failure 101
Benefits of Parallel Sysplex: Capacity and scaling 102
Benefits of Parallel Sysplex: Dynamic workload balancing. 102
Benefits of Parallel Sysplex: Ease of use 103
Benefits of Parallel Sysplex: Single system image 104
Benefits of Parallel Sysplex: Compatible change and non-disruptive growth 105
Benefits of Parallel Sysplex: Application compatibility 105
Benefits of Parallel Sysplex: Disaster recovery . . 106

Notices 107
Programming interface information 108
Trademarks 109

© Copyright IBM Corp. 2006, 2008

Introduction to z/OS

As a technical professional, you need a general understanding of the mainframe computer and its place in today's information technology (IT) organization. **z/OS concepts** is a collection of articles that provides the background knowledge and skills necessary to begin using the basic facilities of a mainframe computer.

For optimal learning, readers are assumed to have successfully completed an introductory course in computer system concepts, such as computer organization and architecture, operating systems, data management, or data communications. They should also have successfully completed courses in one or more programming languages, and be PC literate.

Readers who will benefit from this collection of articles include data processing professionals who have experience with non-mainframe platforms, or who are familiar with some aspects of the mainframe but want to become knowledgeable with other facilities and benefits of the mainframe environment. After reading this collection, readers will have:

- A comprehensive overview of z/OS, a widely used mainframe operating system
- An understanding of mainframe workloads and an overview of the major middleware applications in use on mainframes today
- The basis for subsequent course work in more advanced, specialized areas of z/OS, such as system administration or application programming

© Copyright IBM Corp. 2006, 2008

Chapter 1. z/OS operating system: Providing virtual environments since the 1960s

z/OS® is known for its ability to serve thousands of users concurrently and for processing very large workloads in a secure, reliable, and expedient manner. Its use of multiprogramming and multiprocessing, and its ability to access and manage enormous amounts of virtual and physical storage as well as I/O operations, makes it ideally suited for running mainframe workloads.

The concept of virtual storage is central to z/OS. Virtual storage is an illusion created by the architecture, in that the system seems to have more storage than it really has. Virtual storage is created through the use of tables to map virtual storage pages to frames in central storage or slots in auxiliary storage. Only those portions of a program that are needed are actually loaded into central storage. z/OS keeps the inactive pieces of address spaces in auxiliary storage.

z/OS is structured around address spaces, which are ranges of addresses in virtual storage. Each user of z/OS gets an address space containing the same range of storage addresses. The use of address spaces in z/OS allows for isolation of private areas in different address spaces for system security, yet also allows for inter-address space sharing of programs and data through a common area accessible to every address space.

In common usage, the terms central storage, real storage, real memory, and main storage are used interchangeably. Likewise, virtual memory and virtual storage are synonymous. The amount of central storage needed to support the virtual storage in an address space depends on the working set of the application being used, and this varies over time. A user does not automatically have access to all the virtual storage in the address space. Requests to use a range of virtual storage are checked for size limitations and then the necessary paging table entries are constructed to create the requested virtual storage. Programs running on z/OS and System z™ mainframes can run with 24-, 31-, or 64-bit addressing (and can switch between these modes if needed). Programs can use a mixture of instructions with 16-bit, 32-bit, or 64-bit operands, and can switch between these if needed.

Mainframe operating systems seldom provide complete operational environments. They depend on licensed programs for middleware and other functions. Many vendors, including IBM®, provide middleware and various utility products. Middleware is a relatively recent term that can embody several concepts at the same time. A common characteristic of middleware is that it provides a programming interface, and applications are written (or partially written) to this interface.

Hardware resources used by z/OS

Mainframe hardware consists of processors and a multitude of peripheral devices such as disk drives (called direct access storage devices or **DASD**), magnetic tape drives, and various types of user consoles.

Tape and DASD are used for system functions and by user programs executed by z/OS.

© Copyright IBM Corp. 2006, 2008

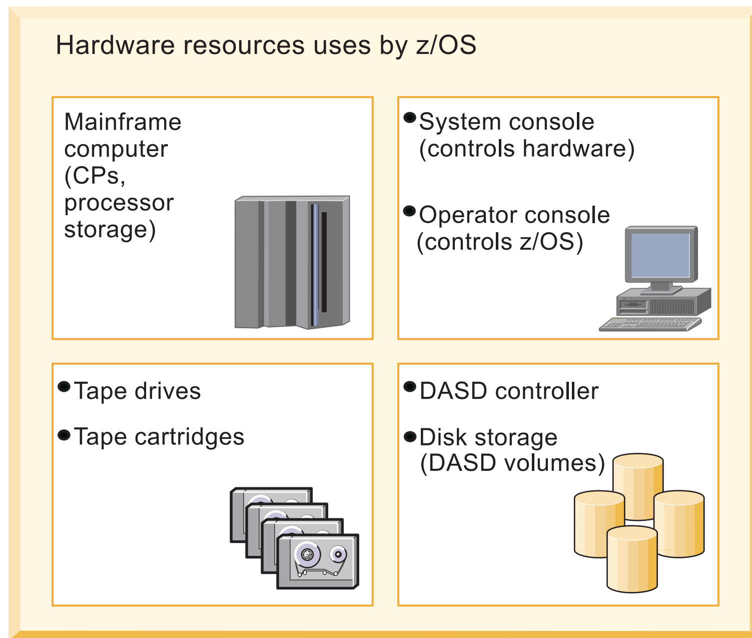


Figure 1. Hardware resources used by z/OS

The z/OS operating system executes in a processor and resides in processor storage during execution. z/OS is commonly referred to as the **system** software.

To fulfill a new order for a z/OS system, IBM ships the system code to the customer through the Internet or (depending on customer preference) on physical tape cartridges. At the customer site, a person such as the z/OS system programmer receives the order and copies the new system to DASD volumes. After the system is customized and ready for operation, system consoles are required to start and operate the z/OS system.

The z/OS operating system is designed to make full use of the latest IBM mainframe hardware and its many sophisticated peripheral devices. Figure 1 presents a simplified view of these mainframe concepts:

Software
: The z/OS operating system consists of load modules or **executable code**. During the install process, the system programmer copies these load modules to **load libraries** residing on DASD volumes.

Hardware
: The system hardware consists of all the devices, controllers, and processors that constitute a mainframe environment.

Peripheral devices
: These include tape drives, DASD, consoles, and many other types of devices.

Processor storage
: Often called real or central storage (or memory), this is where the z/OS operating system executes. Also, all user programs share the use of processor storage with the operating system.

As a "Big Picture" of a typical mainframe hardware configuration, Figure 1 is far from complete. Not shown, for example, are the hardware control units that connect the mainframe to the other tape drives, DASD, and consoles.

Multiprogramming and multiprocessing

z/OS is capable of **multiprogramming**, or executing many programs concurrently, and of **multiprocessing**, which is the simultaneous operation of two or more processors that share the various hardware resources.

The earliest operating systems were used to control single-user computer systems. In those days, the operating system would read in one job, find the data and devices the job needed, let the job run to completion, and then read in another job. In contrast, the computer systems that z/OS manages are capable of **multiprogramming**, or executing many programs concurrently. With multiprogramming, when a job cannot use the processor, the system can suspend, or **interrupt**, the job, freeing the processor to work on another job.

z/OS makes multiprogramming possible by capturing and saving all the relevant information about the interrupted program before allowing another program to execute. When the interrupted program is ready to begin executing again, it can resume execution just where it left off. Multiprogramming allows z/OS to run thousands of programs simultaneously for users who might be working on different projects at different physical locations around the world.

z/OS can also perform **multiprocessing**, which is the simultaneous operation of two or more processors that share the various hardware resources, such as memory and external disk storage devices.

The techniques of multiprogramming and multiprocessing make z/OS ideally suited for processing workloads that require many input/output (I/O) operations. Typical mainframe workloads include long-running applications that write updates to millions of records in a database, and online applications for thousands of interactive users at any given time.

By way of contrast, consider the operating system that might be used for a single-user computer system. Such an operating system would need to execute programs on behalf of one user only. In the case of a personal computer (PC), for example, the entire resources of the machine are often at the disposal of one user.

Many users running many separate programs means that, along with large amounts of complex hardware, z/OS needs large amounts of memory to ensure suitable system performance. Large companies run sophisticated business applications that access large databases and industry-strength middleware products. Such applications require the operating system to protect privacy among users, as well as enable the sharing of databases and software services.

Thus, multiprogramming, multiprocessing, and the need for a large amount of memory mean that z/OS must provide function beyond simple, single-user applications. The related concepts listed below explain the attributes that enable z/OS to manage complex computer configurations.

z/OS programming constructs: Modules, macros, components and control blocks

z/OS is made up of programming instructions that control the operation of the computer system.

These instructions ensure that the computer hardware is being used efficiently and is allowing application programs to run. z/OS includes sets of instructions that, for example, accept work, convert work to a form that the computer can recognize, keep track of work, allocate resources for work, execute work, monitor work, and handle output. A group of related instructions is called a **routine** or **module**. A set of related modules that make a particular system function possible is called a **system component**. The workload management (WLM) component of z/OS, for instance, controls system resources, while the recovery termination manager (RTM) handles system recovery.

Sequences of instructions that perform frequently used system functions can be invoked with executable macro instructions, or **macros**. z/OS macros exist for functions such as opening and closing data files, loading and deleting programs, and sending messages to the computer operator.

As programs execute the work of a z/OS system, they keep track of this work in storage areas known as **control block**. In general, there are four types of z/OS control blocks:

- System-related control blocks
- Resource-related control blocks
- Job-related control blocks
- Task-related control blocks

Each system-related control block represents one z/OS system and contains system-wide information, such as how many processors are in use. Each resource-related control block represents one resource, such as a processor or storage device. Each job-related control block represents one job executing on the system. Each task-related control block represents one unit of work.

Control blocks serve as vehicles for communication throughout z/OS. Such communication is possible because the structure of a control block is known to the programs that use it, and thus these programs can find needed information about the unit of work or resource. 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. The operating system can search the queue to find information about 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)

z/OS uses a huge variety of control blocks, many with very specialized purposes. The three most commonly used control blocks are:

- Task control block (TCB), which represents a unit of work or **task**
- Service request block (SRB), which represents a request for a system service
- Address space control block (ASCB), which represents an address space

Physical storage used by z/OS

Conceptually, mainframes and all other computers have two types of physical storage: Internal and external.

- Physical storage located on the mainframe processor itself. This is called processor storage, real storage or **central storage**; think of it as **memory** for the mainframe.

- Physical storage external to the mainframe, including storage on direct access devices, such as disk drives and tape drives. This storage is called paging storage or **auxiliary storage**.

The primary difference between the two kinds of storage relates to the way in which it is accessed, as follows:

- Central storage is accessed synchronously with the processor. That is, the processor must wait while data is retrieved from central storage.
- Auxiliary storage is accessed asynchronously. The processor accesses auxiliary storage through an input/output (I/O) request, which is scheduled to run amid other work requests in the system. During an I/O request, the processor is free to execute other, unrelated work.

As with memory for a personal computer, mainframe central storage is tightly coupled with the processor itself, whereas mainframe auxiliary storage is located on (comparatively) slower, external disk and tape drives. Because central storage is more closely integrated with the processor, it takes the processor much less time to access data from central storage than from auxiliary storage. Auxiliary storage, however, is less expensive than central storage. Most z/OS installations use large amounts of both.

What is virtual storage?

z/OS uses both types of physical storage (central and auxiliary) to enable another kind of storage called **virtual storage**. In z/OS, each user has access to virtual storage, rather than physical storage. This use of virtual storage is central to the unique ability of z/OS to interact with large numbers of users concurrently, while processing the largest workloads.

Virtual storage means that each running program can assume it has access to all of the storage defined by the architecture's addressing scheme. 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 central storage for the processor to access them.

z/OS supports 64-bit long addresses, which allows a program to address up to 18,446,744,073,709,600,000 bytes (16 exabytes) of storage locations. In reality, the mainframe might have **much less** central storage installed. How much less depends on the model of the computer and the system configuration.

To allow each user to act as though this much storage really exists in the computer system, z/OS keeps only the active portions of each program in central storage. It keeps the rest of the code and data in files called **page data sets** on auxiliary storage, which usually consists of a number of high-speed direct access storage devices (DASD).

Virtual storage, then, is this combination of real and auxiliary storage. z/OS uses a series of tables and indexes to relate locations on auxiliary storage to locations in central storage. It uses special settings (bit settings) to keep track of the identity and authority of each user or program. z/OS uses a variety of storage manager components to manage virtual storage.

Mainframe workers use the terms central storage, real memory, real storage, and main storage interchangeably. Likewise, they use the terms virtual memory and virtual storage synonymously.

What is an address space?

The range of virtual addresses that the operating system assigns to a user or separately running program is called an **address space**. This is the area of contiguous virtual addresses available for executing instructions and storing data.

The range of virtual addresses in an address space starts at zero and can extend to the highest address permitted by the operating system architecture.

z/OS provides each user with a unique address space and maintains the distinction between the programs and data belonging to each address space. Within each address space, the user can start multiple tasks, using task control blocks or TCBs that allow multiprogramming.

In some ways a z/OS address space is like a UNIX® process, and the address space identifier (ASID) is like a process ID (PID). Further, TCBs are like UNIX threads in that each operating system supports processing multiple instances of work concurrently.

However, the use of multiple virtual address spaces in z/OS holds some special advantages. Virtual addressing permits an addressing range that is greater than the central storage capabilities of the system. The use of multiple virtual address spaces provides this virtual addressing capability to each job in the system by assigning each job its own separate virtual address space. The potentially large number of address spaces provides the system with a large virtual addressing capacity.

With multiple virtual address spaces, errors are confined to one address space, except for errors in commonly addressable storage, thus improving system reliability and making error recovery easier. Programs in separate address spaces are protected from each other. Isolating data in its own address space also protects the data.

z/OS uses many address spaces. There is at least one address space for each job in progress and one address space for each user logged on through TSO, telnet, rlogin or FTP (users logged on z/OS through a major subsystem, such as CICS® or IMS™, are using an address space belonging to the subsystem, not their own address spaces). There are many address spaces for operating system functions, such as operator communication, automation, networking, security, and so on.

The use of address spaces allows z/OS to maintain the distinction between the programs and data belonging to each address space. The private areas in one user's address space are isolated from the private areas in other address spaces, and this **address space isolation** provides much of the operating system's security.

Yet, each address space also contains a common area that is accessible to every other address space. Because it maps all of the available addresses, an address space includes system code and data as well as user code and data. Thus, not all of the mapped addresses are available for user code and data.

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 central storage and code words for protecting data files and programs, separate address spaces ensure that users' programs and data do not overlap.

In a multiple virtual address space environment, applications need ways to communicate between address spaces. z/OS provides two methods of **inter-address space communication**:

- Scheduling a service request block (SRB), an asynchronous process
- Using cross-memory services and access registers, a synchronous process

A program uses an SRB to initiate a process in another address space or in the same address space. The SRB is asynchronous in nature and runs independently of the program that issues it, thereby improving the availability of resources in a multiprocessing environment.

A program uses cross-memory services to access another user's address spaces directly. You might compare z/OS cross-memory services to the UNIX Shared Memory functions, which can be used on UNIX without special authority. Unlike UNIX, however, z/OS cross-memory services require the issuing program to have special authority, controlled by the authorized program facility (APF). This method allows efficient and secure access to data owned by others, data owned by the user but stored in another address space for convenience, and for rapid and secure communication with services like transaction managers and database managers.

What is dynamic address translation?

Dynamic address translation, or DAT, is the process of translating a virtual address during a storage reference into the corresponding real address.

If the virtual address is already in central storage, the DAT process may be accelerated through the use of a translation lookaside buffer. If the virtual address is not in central storage, a page fault interrupt occurs, z/OS is notified and brings the page in from auxiliary storage.

Looking at this process more closely reveals that the machine can present any one of a number of different types of faults. A type, region, segment, or page fault will be presented depending on at which point in the DAT structure invalid entries are found. The faults repeat down the DAT structure until ultimately a page fault is presented and the virtual page is brought into central storage either for the first time (there is no copy on auxiliary storage) or by bringing the page in from auxiliary storage.

DAT is implemented by both hardware and software through the use of page tables, segment tables, region tables and translation lookaside buffers. DAT allows different address spaces to share the same program or other data that is for read only. This is because virtual addresses in different address spaces can be made to translate to the same frame of central storage. Otherwise, there would have to be many copies of the program or data, one for each address space.

How z/OS uses physical and virtual storage

By bringing pieces of the program into central storage only when the processor is ready to execute them, z/OS can execute more and larger programs concurrently.

For a processor to execute a program instruction, both the instruction and the data it references must be in central storage. The convention of early operating systems

was to have the entire program reside in central storage when its instructions were executing. However, the entire program does not really need to be in central storage when an instruction executes. Instead, by bringing pieces of the program into central storage only when the processor is ready to execute them--moving them out to auxiliary storage when it doesn't need them, an operating system can execute more and larger programs concurrently.

How does the operating system keep track of each program piece? How does it know whether it is in central storage or auxiliary storage, and where? It is important for z/OS professionals to understand how the operating system makes this happen.

Physical storage is divided into areas, each the same size and accessible by a unique address. In central storage, these areas are called frames; in auxiliary storage, they are called slots. Similarly, the operating system can divide a program into pieces the size of frames or slots and assign each piece a unique address. This arrangement allows the operating system to keep track of these pieces. In z/OS, the program pieces are called pages.

Pages are referenced by their virtual addresses and not by their real addresses. From the time a program enters the system until it completes, the virtual address of the page remains the same, regardless of whether the page is in central storage or auxiliary storage. Each page consists of individual locations called bytes, each of which has a unique virtual address.

How virtual storage addressing works in z/OS

z/OS manages address spaces in units of various sizes; these units are tracked in a virtual address.

Virtual storage is an illusion created by the architecture, in that the system seems to have more memory than it really has. Each user or program gets an address space, and each address space contains the same range of storage addresses. Only those portions of the address space that are needed at any point in time are actually loaded into central storage. z/OS keeps the inactive pieces of address spaces in auxiliary storage. z/OS manages address spaces in units of various sizes, as follows:

- Page address spaces are divided into 4-kilobyte units of virtual storage called pages.
- Segment address spaces are divided into 1-megabyte units called segments. A segment is a block of sequential virtual addresses spanning megabytes, beginning at a 1-megabyte boundary. A 2-gigabyte address space, for example, consists of 2048 segments.
- Region address spaces are divided into 2-8 gigabyte units called regions. A region is a block of sequential virtual addresses spanning 2-8 gigabytes, beginning at a 2-gigabyte boundary. A 2-terabyte address space, for example, consists of 2048 regions.
- A virtual address, accordingly, is divided into four principal fields: Bits 0-32 are called the region index (RX), bits 33-43 are called the segment index (SX), bits 44-51 are called the page index (PX), and bits 52-63 are called the byte index (BX).

A virtual address has the format shown in Figure 2 on page 9.

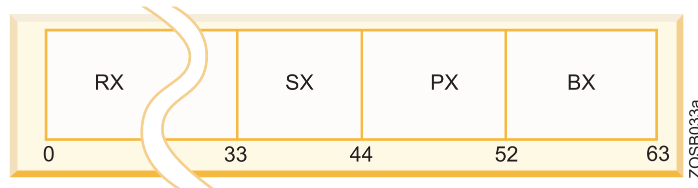


Figure 2. Virtual address format

As determined by its address-space-control element, a virtual address space can be a 2-gigabyte space consisting of one region, or as large as a 16-exabyte space. The RX part of a virtual address for a 2-gigabyte address space must be all zeros; otherwise, an exception is recognized.

The RX part of a virtual address is itself divided into three fields. Bits 0-10 are called the region first index (RFX), bits 11-21 are called the region second index (RSX), and bits 22-32 are called the region third index (RTX). Bits 0-32 of the virtual address have the format shown in Figure 3.

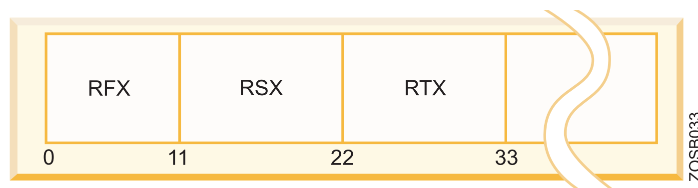


Figure 3. RX field format

A virtual address in which the RTX is the left most significant part (a 42-bit address) is capable of addressing 4 terabytes (4096 regions), one in which the RSX is the left most significant part (a 53-bit address) is capable of addressing 8 petabytes (four million regions), and one in which the RFX is the left most significant part (a 64-bit address) is capable of addressing 16 exabytes (8 billion regions).

What is paging?

When a program is selected for execution, the system brings it into virtual storage, divides it into pages of four kilobytes, transfers the pages into central storage for execution. To the programmer, the entire program appears to occupy contiguous space in storage at all times. Actually, not all pages of a program are necessarily in central storage, and the pages that are in central storage do not necessarily occupy contiguous space.

The pieces of a program executing in virtual storage must be moved between real and auxiliary storage. To allow this, z/OS manages storage in units, or **blocks**, of four kilobytes. The following blocks are defined:

- A block of central 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: Four kilobytes. An active virtual storage page resides in a central storage frame. A virtual storage page that becomes inactive resides in an auxiliary storage slot (in a paging data set). Figure 4 shows the relationship of pages, frames, and slots.

In Figure 4, z/OS is performing paging for a program running in virtual storage. The lettered boxes represent parts of the program. In this simplified view, program parts A, E, F, and H are active and running in central storage frames, while parts B, C, D, and G are inactive and have been moved to auxiliary storage slots. All of the program parts, however, reside in virtual storage and have virtual storage addresses.

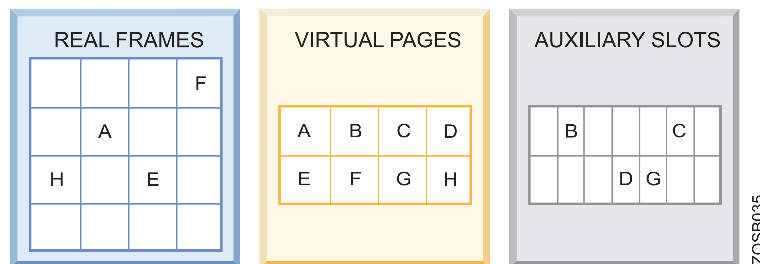


Figure 4. Frames, pages, and slots

z/OS uses a series of tables to determine whether a page is in real or auxiliary storage, and where. To find a page of a program, z/OS checks the table for the virtual address of the page, rather than searching through all of physical storage for it. z/OS then transfers the page into central storage or out to auxiliary storage as needed. This movement of pages between auxiliary storage slots and central storage frames is called **paging**. Paging is key to understanding the use of virtual storage in z/OS.

z/OS paging is transparent to the user. During job execution, only those pieces of the application that are required are brought in, or **paged in**, to central storage. The pages remain in central storage until no longer needed, or until another page is required by the same application or a higher-priority application and no empty central storage is available. To select pages for paging out to auxiliary storage, z/OS follows a ″Least Used″ algorithm. That is, z/OS assumes that a page that has not been used for some time will probably not be used in the near future.

How paging works in z/OS

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.

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 central storage frame. To resolve this page fault, the system must bring the page in from auxiliary storage. First, however, it must locate an available central storage frame. If none is available, the request must be saved and an assigned frame 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.

After a frame is located for the required page, the contents of the page are copied from auxiliary storage to central storage and the page table invalid bit is set off. This operation is called a page-in.

Paging can also take place when z/OS loads an entire program into virtual storage. z/OS obtains virtual storage for the user program and allocates a central 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 central storage while the pages not currently active might be paged out to auxiliary storage.

z/OS tries to keep an adequate supply of available central storage frames on hand. When a program refers to a page that is not in central storage, z/OS uses a central storage page frame from a supply of available frames.

When this supply becomes low, z/OS uses **page stealing** to replenish it, that is, it 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 central storage frame. Pages that have not been active for a relatively long time are good candidates for page stealing.

z/OS uses a sophisticated paging algorithm to efficiently manage virtual storage based on which pages were most recently used. An **unreferenced interval count** indicates how long it has been since a program referenced the page. At regular intervals, the system checks the reference bit for each page frame. If the reference bit is off– that is, the frame has not been referenced– the system adds to the 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 frame to zero. Frames with the highest unreferenced interval counts are the ones most likely to be stolen.

Swapping and the working set

Swapping is the process of transferring all of the pages of an address space between central storage and auxiliary storage.

A swapped-in address space is active, having pages in central storage frames and pages in auxiliary storage slots. A swapped-out address space is inactive; the address space resides on auxiliary storage and cannot execute until it is swapped in.

While only a subset of the address space's pages (known as its **working set**) would likely be in central storage at any time, swapping effectively moves the entire address space. It is one of several methods that z/OS uses to balance the system workload and ensure that an adequate supply of available central storage frames is maintained.

Swapping is performed by the System Resource Manager (SRM) component, in response to recommendations from the Workload Manager (WLM) component.

What is storage protection?

Many programs and users are competing for the use of the system. So how does z/OS preserves the integrity of each user's work? One technique is through the use of multiple storage protect keys.

Under z/OS, the information in central 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 central storage.

When a request is made to modify the contents of a central storage location, the key associated with the request is compared to the storage protect key. 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 central 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 central 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.

z/OS uses 16 storage protect keys. A specific key is assigned according to the type of work being performed. The key is stored in bits 8 through 11 of the program status word (PSW). A PSW is assigned to each job in the system.

Storage protect keys 0 through 7 are used by the z/OS base control program (BCP) and various subsystems and middleware products. Storage protect key 0 is the master key. Its use is restricted to those parts of the BCP that require almost unlimited store and fetch capabilities. In almost any situation, a storage protect key of 0 associated with a request to access or modify the contents of a central storage location means that the request will be satisfied.

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.

The role of storage managers

Central storage frames and auxiliary storage slots, and the virtual storage pages that they support, are managed by separate components of z/OS. These components are known as the real storage manager (not **central** storage manager), the auxiliary storage manager, and the virtual storage manager.

Real storage manager

The **real storage manager** or **RSM** keeps track of the contents of central storage. It manages the paging activities– such as page-in, page-out, and page stealing– and helps with swapping an address space in or out. RSM also performs **page fixing**, which is marking pages as unavailable for stealing.

Auxiliary storage manager

The **auxiliary storage manager** or **ASM** uses the system's page data sets to keep track of auxiliary storage slots. Specifically:

- Slots for virtual storage pages that are not in central storage frames
- Slots for pages that do not occupy frames but, because the frame's contents have not been changed, the slots are still valid.

When a page-in or page-out is required, ASM works with RSM to locate the proper central storage frames and auxiliary storage slots.

Virtual storage manager

The **virtual storage manager** or **VSM** responds to requests to obtain and free virtual storage. VSM also manages storage allocation for any program that must run in real, rather than virtual storage. Real storage is allocated to code and data when they are loaded in virtual storage. As they run, programs can request more storage by means of a system service, such as the GETMAIN macro. Programs can release storage with the FREEMAIN macro.

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**, which are logically related areas of virtual storage identified by the numbers 0 to 255. Being logically related means the storage areas within a subpool share characteristics such as:

- Storage protect key
- Whether they are fetch protected, pageable, or swappable
- Where they must reside in virtual storage (above or below 16 megabytes)
- Whether they can be shared by more than one task

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

A brief history of virtual storage and 64-bit addressability

In 1970, IBM introduced System/370™, the first of its architectures to use virtual storage and address spaces. Since that time, the operating system has changed in many ways. One key area of growth and change is addressability.

A program running in an address space can reference all of the storage associated with that address space. A program's ability to reference all of the storage associated with an address space is called **addressability**.

System/370 defined storage addresses as 24 bits in length, which meant that the highest accessible address was 16,777,215 bytes (or 224-1 bytes). The use of 24-bit addressability allowed MVS/370, the operating system at that time, to allot to each user an address space of 16 megabytes. Over the years, as MVS/370 gained more functions and was asked to handle more complex applications, even access to 16 megabytes of virtual storage fell short of user needs.

With the release of the System/370-XA architecture in 1983, IBM extended the addressability of the architecture to 31 bits. With 31-bit addressing, the operating system (now called MVS™ Extended Architecture or MVS/XA™) increased the addressability of virtual storage from 16 MB to 2 gigabytes (2 GB). In other words, MVS/XA provided an address space for users that was 128 times larger than the address space provided by MVS/370. The 16 MB address became the dividing point between the two architectures and is commonly called the **line** (see Figure 5 on page 14).

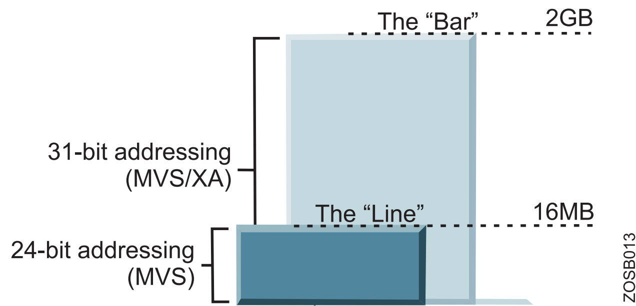


Figure 5. 31-bit addressability allows for 2-gigabyte address spaces in MVS/XA

The new architecture did not require customers to change existing application programs. To maintain compatibility for existing programs, MVS/XA remained compatible for programs originally designed to run with 24-bit addressing on MVS/370, while allowing application developers to write new programs to exploit the 31-bit technology.

To preserve compatibility between the different addressing schemes, MVS/XA did not use the **high-order bit** of the address (Bit 0) for addressing. Instead, MVS/XA reserved this bit to indicate how many bits would be used to resolve an address: 31-bit addressing (Bit 0 on) or 24-bit addressing (Bit 0 off).

With the release of zSeries® mainframes in 2000, IBM further extended the addressability of the architecture to 64 bits. With 64-bit addressing, the potential size of a z/OS address space expands to a size so vast we need new terms to describe it. Each address space, called a 64-bit address space, is 16 exabytes (EB) in size; an exabyte is slightly more than one billion gigabytes. The new address space has logically 264 addresses. It is 8 billion times the size of the former 2-gigabyte address space, or 18,446,744,073,709,600,000 bytes (Figure 6 on page 15).

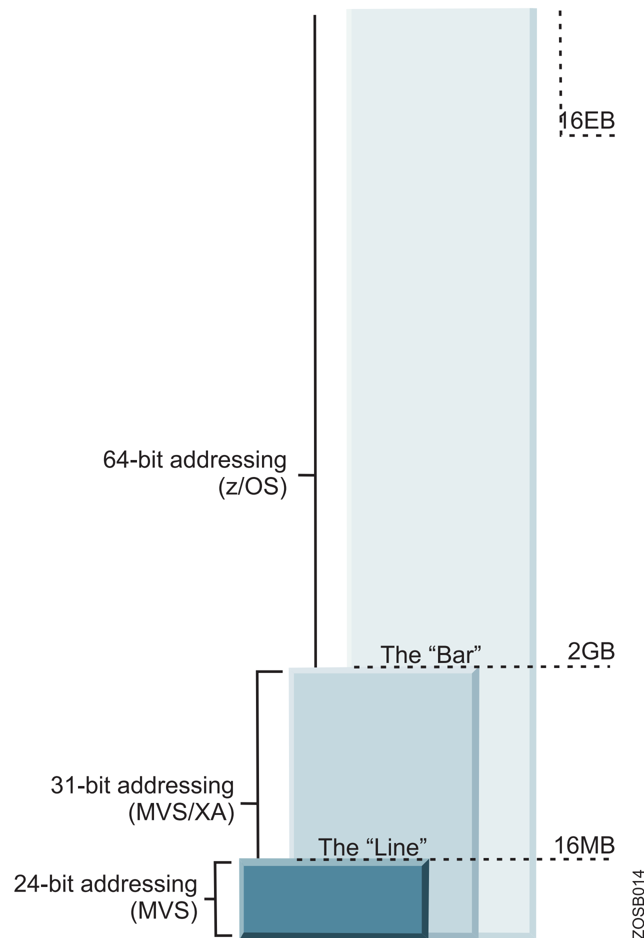


Figure 6. 64-bit addressability allows for 16 exabytes of addressable storage

We say that the potential size is 16 exabytes because z/OS, by default, continues to create address spaces with a size of 2 gigabytes. The address space exceeds this limit only if a program running in it allocates virtual storage above the 2-gigabyte address. If so, z/OS increases the storage available to the user from two gigabytes to 16 exabytes.

A program running on z/OS and the zSeries mainframe can run with 24-, 31-, or 64-bit addressing (and can switch among these if needed). To address the high virtual storage available with the 64-bit architecture, the program uses 64-bit-specific instructions. Although the architecture introduces unique 64-bit exploitation instructions, the program can use both 31-bit and 64-bit instructions, as needed.

For compatibility, the layout of the storage areas for an address space is the same below 2 gigabytes, providing an environment that can support both 24-bit and 31-bit addressing. The area that separates the virtual storage area below the 2-gigabyte address from the user private area is called the **bar**, as shown in Figure 7 on page 16. The user private area is allocated for application code rather than operating system code.

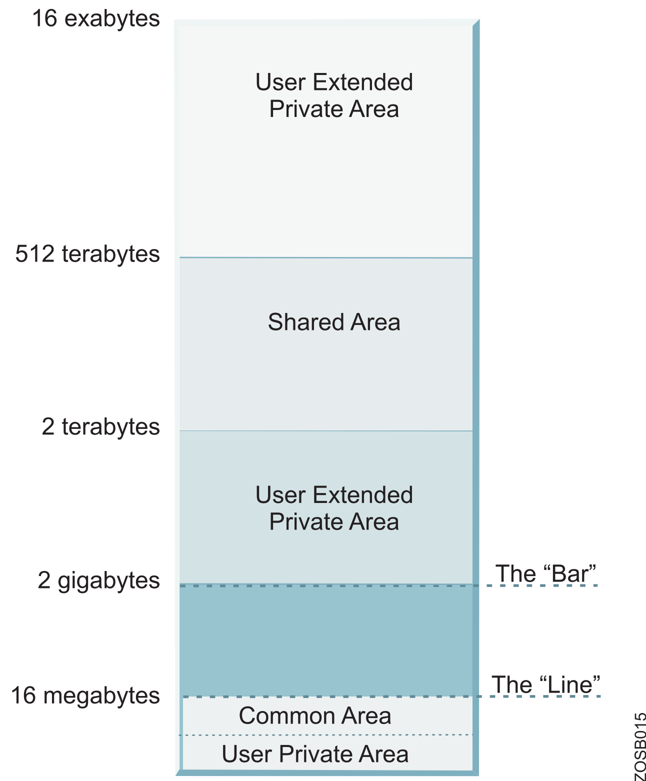


Figure 7. Storage map for a 64-bit address space

0 to 2^{31}
: The layout is the same; see Figure 7.

2^{31} to 2^{32}
: From 2 GB to 4 GB is considered the bar. Below the bar can be addressed with a 31-bit address. Above the bar requires a 64-bit address.

2^{32} - 2^{41}
: The low non-shared area (user private area) starts at 4 GB and extends to 2^{41} .

2^{41} - 2^{50}
: Shared area (for storage sharing) starts at 2^{41} and extends to 2^{50} or higher, if requested.

2^{50} - 2^{64}
: High non-shared area (user private area) starts at 2^{50} or wherever the shared area ends, and goes to 2^{64}.

In a 16-exabyte address space with 64-bit virtual storage addressing, there are three additional levels of translation tables, called region tables: Region third table (R3T), region second table (R2T), and region first table (R1T). The region tables are 16 KB in length, and there are 2048 entries per table. Each region has 2 GB.

Segment tables and page table formats remain the same as for virtual addresses below the bar. When translating a 64-bit virtual address, once the system has identified the corresponding 2-GB region entry that points to the segment table, the process is the same as that described previously.

What is meant by “below-the-line” storage?

z/OS programs and data reside in virtual storage that, when necessary, is backed by central storage. Most programs and data do not depend on their real addresses. Some z/OS programs, however, do depend on real addresses and some require these real addresses to be less than 16 megabytes. z/OS programmers refer to this storage as being ″below the 16-megabyte line.″

In z/OS, a program’s attributes include one called **residence mode** or RMODE, which specifies whether the program must reside (be loaded) in storage below 16 megabytes. A program with RMODE(24) must reside below 16 megabytes, while a program with RMODE(31) can reside anywhere in virtual storage.

Examples of programs that require below-the-line storage include any program that allocates a data control block (DCB). Those programs, however, often can be 31-bit residency mode or RMODE(31) as they can run in 31-bit addressing mode or AMODE(31). z/OS reserves as much central storage below 16 megabytes as it can for such programs and, for the most part, handles their central storage dependencies without requiring them to make any changes.

Thousands of programs in use today are AMODE(24) and therefore RMODE(24). Every program written before MVS/XA was available, and not subsequently changed, has that characteristic. There are relatively few reasons these days why a new program might need to be AMODE(24), so a new application likely has next to nothing that is RMODE(24).

What’s in an address space?

One way of thinking of an address space is as a programmer’s map of the virtual storage available for code and data. An address space provides each programmer with access to all of the addresses available through the computer architecture.

z/OS provides each user with a unique address space and maintains the distinction between the programs and data belonging to each address space. Because it maps all of the available addresses, however, an address space includes system code and data as well as user code and data. Thus, not all of the mapped addresses are available for user code and data.

Understanding the division of storage areas in an address space is made easier with a diagram. The diagram shown in Storage areas in an address space shows how an address space maintains the distinction between programs and data belonging to the user, and those belonging to the operating system.

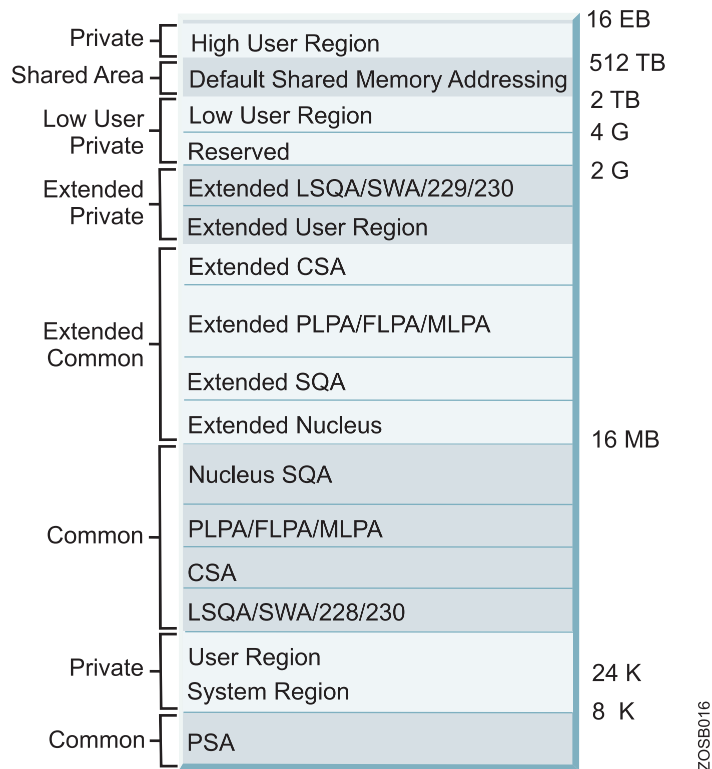


Figure 8. Storage areas in an address space

Figure 8 shows the major storage areas in each address space. These are described briefly as follows:

All storage above 2 GB
This area is called **high virtual storage** and is addressable only by programs running in 64-bit mode. It is divided by the high virtual shared area, which is an area of installation-defined size that can be used to establish cross-address space viewable connections to obtained areas within this area.

Extended areas above 16 MB
This range of areas, which lies above the 16 MB line but below the 2 GB bar, is a kind of ″mirror image″ of the common area below 16 MB. They have the same attributes as their equivalent areas below the line, but because of the additional storage above the 16 MB line, their sizes are much larger.

Nucleus
This is a key 0, read-only area of common storage that contains operating system control programs.

SQA This area contains system level (key 0) data accessed by multiple address spaces. The SQA area is not pageable (fixed), which means that it resides in central storage until it is freed by the requesting program. The size of the SQA area is predefined by the installation and cannot change while the operating system is active. Yet it has the unique ability to ″overflow″ into the CSA area as long as there is unused CSA storage that can be converted to SQA.

PLPA/FLPA/MLPA
This area contains the link pack areas (the pageable link pack area, fixed

link pack area, and modified link pack area), which contain system level programs that are often run by multiple address spaces. For this reason, the link pack areas reside in the common area which is addressable by every address space, therefore eliminating the need for each address space to have its own copy of the program. This storage area is below the 16 MB line and is therefore addressable by programs running in 24-bit mode.

CSA This portion of common area storage (addressable by all address spaces) is available to all applications. The CSA is often used to contain data frequently accessed by multiple address spaces. The size of the CSA area is established at system initialization time (IPL) and cannot change while the operating system is active.

LSQA/SWA/subpool 228/subpool 230
This assortment of subpools, each with specific attributes, is used primarily by system functions when the functions require address space level storage isolation. Being below the line, these areas are addressable by programs running in 24-bit mode.

User Region
This area is obtainable by any program running in the user's address space, including user key programs. It resides below the 16 MB line and is therefore addressable by programs running in 24-bit mode.

System Region
This small area (usually only four pages) is reserved for use by the region control task of each address space.

Prefixed Save Area (PSA)
This area is often referred to as "low core." The PSA is a common area of virtual storage from address zero through 8191 in every address space. There is one unique PSA for every processor installed in a system. The PSA maps architecturally fixed hardware and software storage locations for the processor. Because there is a unique PSA for each processor, from the view of a program running on z/OS, the contents of the PSA can change any time the program is dispatched on a different processor. This feature is unique to the PSA area and is accomplished through a unique DAT manipulation technique called **prefixing**.

Given the vast range of addressable storage in an address space, the drawing in Figure 8 on page 18 is not to scale.

Each address space in the system is represented by an address space control block or ASCB. To represent an address space, the system creates an ASCB in common storage (system queue area or SQA), which makes it accessible to other address spaces.

System address spaces and the master scheduler

Many z/OS system functions run in their own address spaces. The master scheduler subsystem, for example, runs in the address space called *MASTER* and is used to establish communication between z/OS and its own address spaces.

When you start z/OS, master initialization routines initialize system services, such as the system log and communication task, and start the master scheduler address space. Then, the master scheduler may start the job entry subsystem (JES2 or JES3). JES is the primary job entry subsystem. On many production systems JES is not started immediately; instead, the automation package starts all tasks in a controlled

sequence. Then other subsystems are started. Subsystems are defined in a special file of system settings called a parameter library or PARMLIB. These subsystems are **secondary subsystems**.

Each address space created has a number associated with it, called the address space ID (or ASID). Because the master scheduler is the first address space created in the system, it becomes address space number 1 (ASID=1). Other system address spaces are then started during the initialization process of z/OS.

At this point, you need only understand that z/OS and its related subsystems require address spaces of their own to provide a functioning operating system. A short description of each type of address space follows:

System
: z/OS system address spaces are started after initialization of the master scheduler. These address spaces perform functions for all the other types of address spaces that start in z/OS.

Subsystem
: z/OS requires the use of various subsystems, such as a primary job entry subsystem or JES. Also, there are address spaces for middleware products such as DB2®, CICS, and IMS.

Besides system address spaces, there are, of course, typically many address spaces for users and separately running programs; for example:

- TSO/E address spaces are created for every user who logs on to z/OS.
- An address space is created for every batch job that runs on z/OS. Batch job address spaces are started by JES.

What is workload management?

For z/OS, the management of system resources is the responsibility of the workload management (WLM) component. WLM manages the processing of workloads in the system according to the company's business goals, such as response time. WLM also manages the use of system resources, such as processors and storage, to accomplish these goals.

In simple terms, WLM has three objectives:

- To achieve the business goals that are defined by the installation, by automatically assigning sysplex resources to workloads based on their importance and goals. This objective is known as **goal achievement**.
- To achieve optimal use of the system resources from the system point of view. This objective is known as **throughput**.
- To achieve optimal use of system resources from the point of view of the individual address space. This objective is known as **response** and **turnaround time**.

Goal achievement is the first and most important task of WLM. Optimizing throughput and minimizing turnaround times of address spaces come after that. Often, these latter two objectives are contradictory. Optimizing throughput means keeping resources busy. Optimizing response and turnaround time, however, requires resources to be available when they are needed. Achieving the goal of an important address space might result in worsening the turnaround time of a less important address space. Thus, WLM must make decisions that represent trade-offs between conflicting objectives.

To balance throughput with response and turnaround time, WLM does the following:

- Monitors the use of resources by the various address spaces.
- Monitors the system-wide use of resources to determine whether they are fully utilized.
- Determines which address spaces to swap out (and when).
- Inhibits the creation of new address spaces or steals pages when certain shortages of central storage exist.
- Changes the dispatching priority of address spaces, which controls the rate at which the address spaces are allowed to consume system resources.
- Selects the devices to be allocated, if a choice of devices exists, to balance the use of I/O devices.

Other z/OS components, transaction managers, and database managers can communicate to WLM a change in status for a particular address space (or for the system as a whole), or to invoke WLM's decision-making power.

For example, WLM is notified when:

- Central storage is configured into or out of the system.
- An address space is to be created.
- An address space is deleted.
- A swap-out starts or completes.
- Allocation routines can choose the devices to be allocated to a request.

Up to this point, we have discussed WLM only in the context of a single z/OS system. In real life, customer installations often use clusters of multiple z/OS systems in concert to process complex workloads. **Parallel Sysplex®** is the term used to refer to clustered z/OS systems.

WLM is particularly well-suited to a sysplex environment. It keeps track of system utilization and workload goal achievement across all the systems in the Parallel Sysplex and data sharing environments. For example, WLM can decide the z/OS system on which a batch job should run, based on the availability of resources to process the job quickly.

A mainframe installation can influence almost all decisions made by WLM by establishing a set of **policies** that allow an installation to closely link system performance to its business needs. Workloads are assigned goals (for example, a target average response time) and an importance (that is, how important it is to the business that a workload meet its goals).

Before the introduction of WLM, the only way to inform z/OS about the company's business goals was for the system programmer to translate from high-level objectives into the extremely technical terms that the system can understand. This translation required highly skilled staff, and could be protracted, error-prone, and eventually in conflict with the original business goals.

Further, it was often difficult to predict the effects of changing a system setting, which might be required, for example, following a system capacity increase. This difficulty could result in unbalanced resource allocation, in which work is deprived of a critical system resource. This way of operating, called compatibility mode, was becoming unmanageable as new workloads were introduced, and as multiple systems were being managed together.

When in goal mode system operation, WLM provides fewer, simpler, and more consistent system externals that reflect goals for work expressed in terms commonly used in business objectives, and WLM and System Resource Manager (SRM) match resources to meet those goals by constantly monitoring and adapting the system. Workload Manager provides a solution for managing workload distribution, workload balancing, and distributing resources to competing workloads.

WLM policies are often based on a service level agreement (SLA), which is a written agreement of the information systems (I/S) service to be provided to the users of a computing installation. WLM tries to achieve the needs of workloads (response time) as described in an SLA by attempting the appropriate distribution of resources without over-committing them. Equally important, WLM maximizes system use (throughput) to deliver maximum benefit from the installed hardware and software platform.

I/O and data management

Nearly all work in the system involves data input or data output. In a mainframe, the channel subsystem manages the use of I/O devices, such as disks, tapes, and printers.

The operating system must associate the data for a given task with a device, and manage file allocation, placement, monitoring, migration, backup, recall, recovery, and deletion.

These data management activities can be done either manually or through the use of automated processes. When data management is automated, the system determines object placement, and automatically manages object backup, movement, space, and security. A typical z/OS production system includes both manual and automated processes for managing data.

Depending on how a z/OS system and its storage devices are configured, a user or program can directly control many aspects of data management, and in the early days of the operating system, users were required to do so. Increasingly, however, z/OS installations rely on installation-specific settings for data and resource management, and add-on storage management products to automate the use of storage. The primary means of managing storage in z/OS is with the Data Facility Storage Management Subsystem (DFSMS™), which comprises a suite of related data and storage management products.

Supervising the execution of work in the system

z/OS uses several types of supervisor controls to enable multiprogramming.

These supervisor controls include:

Interrupt processing

Multiprogramming requires that there be some technique for switching 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 z/OS, this switch is achieved by **interrupts**, which are events that alter the sequence in which the processor executes instructions. When an interrupt occurs, the system saves the execution status of the interrupted routine and analyzes and processes the interrupt.

Creating dispatchable units of work
: To identify and keep track of its work, the z/OS operating system represents each unit of work with a control block. Two types of control blocks represent dispatchable units of work: **Task control blocks** or TCBs represent tasks executing within an address space; **service request blocks** or SRBs represent higher priority system services.

Dispatching work
: After interrupts are processed, the operating system determines which unit of work (of all the units of work in the system) is ready to run and 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 data file), the operating system must prevent other programs from using the resource until the interrupted program has completed its processing of the resource.

  Several techniques exist for serializing the use of resources; **enqueuing** and **locking** are the most common (a third technique is called **latching**). All users can use enqueuing, but only authorized routines can use locking to serialize the use of resources.

What is interrupt processing?

An *interrupt* is an event that alters the sequence in which the processor executes instructions.

An interrupt might be planned (specifically requested by the currently running program) or unplanned (caused by an event that might or might not be related to the currently running program). z/OS uses six types of interrupts, as follows:

Supervisor calls or SVC interrupts
: These interrupts occur when the program issues an SVC to request a particular system service. An SVC interrupts the program being executed and passes control to the supervisor so that it can perform the service. Programs request these services through macros such as OPEN (open a file), GETMAIN (obtain storage), or WTO (write a message to the system operator).

I/O interrupts
: These interrupts occur when the channel subsystem signals a change of status, such as an input/output (I/O) operation completing, an error occurring, or an I/O device such as a printer has become ready for work.

External interrupts
: These interrupts can indicate any of several events, such as a time interval expiring, the operator pressing the interrupt key on the console, or the processor receiving a signal from another processor.

Restart interrupts
: These interrupts 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 interrupts
: These interrupts are caused by program errors (for example, the program

attempts to perform an invalid operation), **page faults** (the program references a page that is not in central storage), or requests to monitor an event.

Machine check interrupts
These interrupts are caused by machine malfunctions.

When an interrupt occurs, the hardware saves pertinent information about the program that was interrupted and, if possible, disables the processor for further interrupts of the same type. The hardware then routes control to the appropriate interrupt handler routine. The program status word or PSW is a key resource in this process.

The program status word (PSW) is a 128-bit data area in the processor that, along with a variety of other types of registers (control registers, timing registers, and 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. Each processor has only one current PSW. Thus, only one task can execute on a processor at a time.

The PSW controls the order in which instructions are fed to the processor, and indicates the status of the system in relation to the currently running program. Although each processor has only one PSW, it is useful to think of three types of PSWs to understand interrupt processing:
- Current PSW
- New PSW
- Old PSW

The current PSW indicates the next instruction to be executed. It also indicates whether the processor is enabled or disabled for I/O interrupts, external interrupts, machine check interrupts, and certain program interrupts. When the processor is enabled, these interrupts can occur. When the processor is disabled, these interrupts are ignored or remain pending.

There is a new PSW and an old PSW associated with each of the six types of interrupts. The new PSW contains the address of the routine that can process its associated interrupt. If the processor is enabled for interrupts when an interrupt occurs, PSWs are switched using the following technique:
1. Storing the current PSW in the old PSW associated with the type of interrupt that occurred.
2. Loading the contents of the new PSW for the type of interrupt that occurred into the current PSW.

The current PSW, which indicates the next instruction to be executed, now contains the address of the appropriate routine to handle the interrupt. This switch has the effect of transferring control to the appropriate interrupt handling routine.

Mainframe architecture provides registers to keep track of things. The PSW, for example, is a register used to contain information that is required for the execution of the currently active program.

Mainframes provide other registers, as follows:

Access registers
These registers specify the address space in which data is found.

General registers
These registers address data in storage, and also hold user data.

Floating point registers
These registers hold numeric data in floating point form.

Control registers
These registers are used by the operating system itself, for example, as references to translation tables.

Figure 9 shows the use of registers and the PSW.

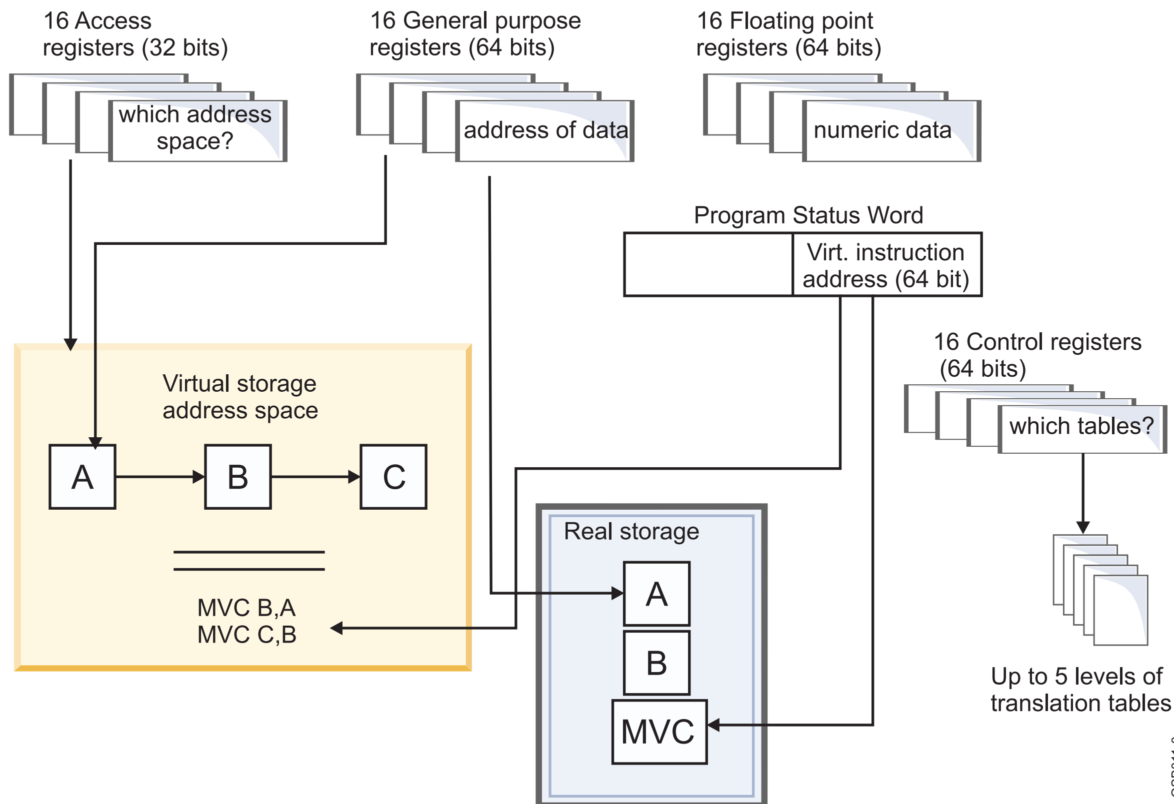


Figure 9. Registers and the PSW

Dispatchable units of work: Tasks and service requests

In z/OS, dispatchable units of work are represented by two kinds of control blocks: Task and service request blocks.

Task control blocks (TCBs)
These control blocks represent tasks executing within an address space, such as user programs and system programs that support the user programs.

A TCB contains information about the running task, such as the address of any storage areas it has created. Do not confuse the z/OS term TCB with the UNIX data structure called a **process control block** or PCB.

TCBs are created in response to an ATTACH macro. By issuing the ATTACH macro, a user program or system routine begins the execution of the program specified on the ATTACH macro, as a subtask of the attacher's task. As a subtask, the specified program can compete for processor time and can use certain resources already allocated to the attacher's task.

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.

Service request blocks (SRBs)

These control blocks represent requests to execute a system 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.

The routine that performs the function or service is called the **SRB routine** ; initiating the process is called **scheduling an SRB**; the SRB routine runs in the operating mode known as **SRB mode**.

An SRB is similar to a TCB in that it identifies a unit of work to the system. Unlike a TCB, an SRB cannot ″own″ storage areas. SRB routines can obtain, reference, use, and free storage areas, but the areas must be owned by a TCB. In a multiprocessor environment, the SRB routine, after being scheduled, can be dispatched on another processor and can run concurrently with the scheduling program. The scheduling program can continue to do other processing in parallel with the SRB routine. As mentioned earlier, an SRB provides a means of asynchronous inter-address space communication for programs running on z/OS.

Only programs running in a mode of higher authority called **supervisor state** can create an SRB. These authorized programs obtain storage and initialize the control block with things such as the identity of the target address space and pointers to the code that will process the request. The program creating the SRB then issues the SCHEDULE macro and indicates whether the SRB has global (system-wide) or local (address space-wide) priority. The system places the SRB on the appropriate dispatching queue where it will remain until it becomes the highest priority work 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 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 interrupts are scheduled at a global priority, to minimize I/O delays.

Preemptable versus non-preemptable units of work

Which routine receives control after an interrupt is processed depends on whether the interrupted unit of work was preemptable.

If so, the operating system determines which unit of work should be performed next. That is, the system determines which unit or work, of all the work in the system, has the highest priority, and passes control to that unit of work.

A non-preemptable unit of work can be interrupted, but must receive control after the interrupt is processed. For example, SRBs are often non-preemptable. Thus, if a routine represented by a non-preemptable SRB is interrupted, it will receive control after the interrupt has been processed. In contrast, a routine represented by a TCB, such as a user program, is usually preemptable. If it is interrupted, control returns to the operating system when the interrupt handling completes. z/OS then determines which task, of all the ready tasks, will execute next.

What does the dispatcher do?

In z/OS, the dispatcher component is responsible for routing control to the highest priority unit of work that is ready to execute.

New work is selected, for example, when a task is interrupted or becomes non-dispatchable, or after an SRB completes or is suspended (that is, an SRB is delayed because a required resource is not available).

The dispatcher processes 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 multiprocessing system fails, alternate CPU recovery is invoked by means of a special exit to recover work that was being executed on the failing processor.
2. SRBs that have a global 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 spaces's priority is determined by the dispatching priority specified by the user or the installation.

   After selecting the highest priority address space, z/OS (through 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, z/OS assumes a state called an **enabled wait** until fresh work enters the system.

Different models of the z/Series hardware can have from one to 54 central processors (CPs). Each and every CP can be executing instructions at the same time. Dispatching priorities determine when ready-to-execute address spaces get dispatched.

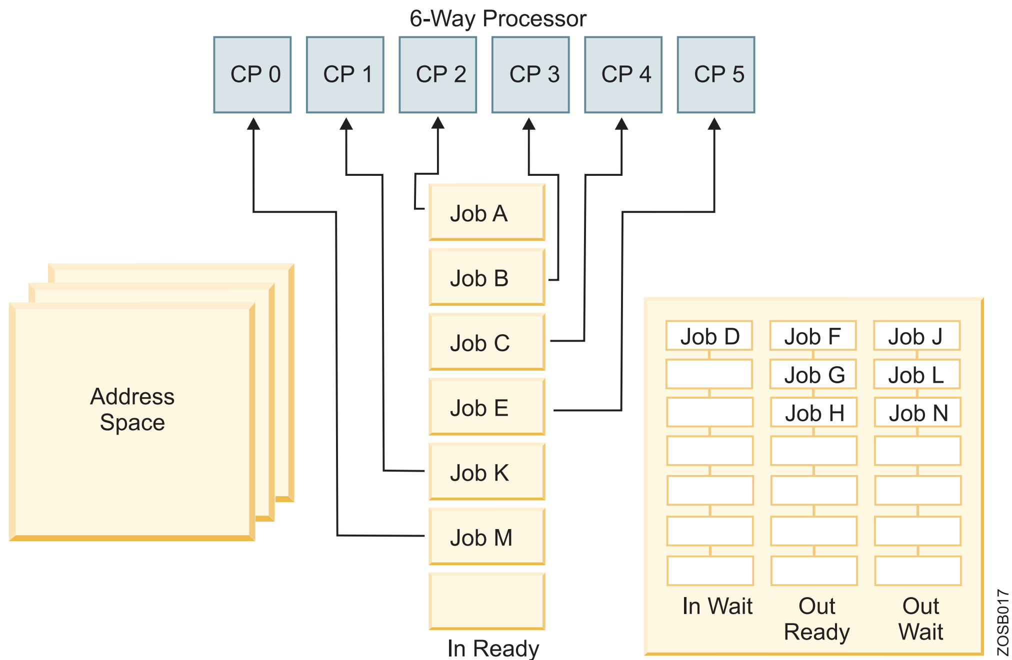


Figure 10. Dispatching work

An address space can be in any one of four queues:

- IN-READY - In central storage and waiting to be dispatched
- IN-WAIT - In central storage but waiting for some event to complete
- OUT-READY - Ready to execute but swapped out
- OUT-WAIT - Swapped out and waiting for some event to complete

Only IN-READY work can be selected for dispatching.

Serializing the use of resources

In a multitasking, multiprocessing environment, resource serialization is the technique used to coordinate access to resources that are used by more than one application.

Programs that change data need exclusive access to the data. Otherwise, if several programs were to update the same data at the same time, the data could be corrupted (also referred to as a loss of data integrity). On the other hand, programs that need only to read data can safely share access to the same data at the same time.

The most common techniques for serializing the use of resources are enqueuing and locking. These techniques allow for orderly access to system resources needed by more than one user in a multiprogramming or multiprocessing environment. In z/OS, enqueuing is managed by the global resource serialization component and locking is managed by various lock manager programs in the supervisor component.

Global resource serialization

The global resource serialization component processes requests for resources from programs running on z/OS. Global resource serialization serializes access to resources to protect their integrity. An installation can

connect two or more z/OS systems with channel-to-channel (CTC) adapters to form a GRS complex to serialize access to resources shared among the systems.

When a program requests access to a reusable resource, the access can be requested as exclusive or shared. When global resource serialization grants shared access to a resource, exclusive users cannot obtain access to the resource. Likewise, when global resource serialization grants exclusive access to a resource, all other requestors for the resource wait until the exclusive requestor frees the resource.

Enqueuing

Enqueuing is the means by which a program running on z/OS requests control of a serially reusable resource. Enqueuing is accomplished by means of the ENQ (enqueue) and DEQ (dequeue) macros, which are available to all programs running on the system. For devices that are shared between multiple z/OS systems, enqueuing is accomplished through the RESERVE and DEQ macros.

On ENQ and RESERVE, a program specifies the names of one or more resources and requests shared or exclusive control of those resources. If the resources are to be modified, the program must request exclusive control; if the resources are not to be modified, the program **should** request shared control, which allows the resource to be shared by other programs that do not require exclusive control. If the resource is not available, the system suspends the requesting program until the resource becomes available. When the program no longer requires control of a resource, the program uses the DEQ macro to release it.

Locking

Through locking, the system serializes the use of system resources by authorized routines and, in a Parallel Sysplex, 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 z/OS, 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 nonreusable or nonsharable routines and various resources.

To use a resource protected by a lock, a routine must first request the lock for that resource. If the lock is unavailable (that is, it is already held by another 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. Most global locks are spin locks. The holder of a spin lock should be disabled for most interrupts (if the holder were to be interrupted, it might never be able to gain control to give up the lock).
- 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. All local locks are suspend locks.

You might wonder what would happen if two users each request a lock that is held by the other? Would they both wait forever for the other to

release the lock first, in a kind of stalemate? In z/OS, such an occurrence would be known as a **deadlock**. Fortunately, the z/OS locking methodology prevents deadlocks.

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 must be acquired in hierarchical sequence. Assume, in this example, that lock A precedes lock B is 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 this hierarchy, a deadlock cannot occur.

Defining characteristics of z/OS

z/OS has several characteristics that distinguish it from other mainframe operating systems.

The defining characteristics of z/OS are summarized as follows:

- The use of address spaces in z/OS holds many advantages: Isolation of private areas in different address spaces provides for system security, yet each address space also provides a common area that is accessible to every address space.
- The system is designed to preserve data integrity, regardless of how large the user population might be. z/OS prevents users from accessing or changing any objects on the system, including user data, except by the system-provided interfaces that enforce authority rules.
- The system is designed to manage a large number of concurrent batch jobs, with no need for the customer to externally manage workload balancing or integrity problems that might otherwise occur due to simultaneous and conflicting use of a given set of data.
- The security design extends to system functions as well as simple files. Security can be incorporated into applications, resources, and user profiles.
- The system allows multiple communications subsystems at the same time, permitting unusual flexibility in running disparate communications-oriented applications (with mixtures of test, production, and fall-back versions of each) at the same time. For example, multiple TCP/IP stacks can be operational at the same time, each with different IP addresses and serving different applications.
- The system provides extensive software recovery levels, making unplanned system restarts very rare in a production environment. System interfaces allow application programs to provide their own layers of recovery. These interfaces are seldom used by simple applications– they are normally used by sophisticated applications.
- The system is designed to routinely manage very disparate workloads, with automatic balancing of resources to meet production requirements established by the system administrator.
- The system is designed to routinely manage large I/O configurations that might extend to thousands of disk drives, multiple automated tape libraries, many large printers, large networks of terminals, and so forth.
- The system is controlled from one or more operator terminals, or from application programming interfaces (APIs) that allow automation of routine operator functions.

- The operator interface is a critical function of z/OS. It provides status information, messages for exception situations, control of job flow, hardware device control, and allows the operator to manage unusual recovery situations.

Additional software products for z/OS

A z/OS system usually contains additional, priced products that are needed to create a practical working system.

For example, a production z/OS system usually includes a security manager product and a database manager product. When talking about z/OS, people often assume the inclusion of these additional products. This assumption is normally apparent from the context of a discussion, but it might sometimes be necessary to ask whether a particular function is part of ″the base z/OS″ or whether it is an add-on product. IBM refers to its own add-on products as **IBM licensed programs**.

With a multitude of **independent software vendors** (ISVs) offering a large number of products with varying but similar functionality, such as security managers and database managers, the ability to choose from a variety of licensed programs to accomplish a task considerably increases the flexibility of the z/OS operating system and allows the mainframe IT group to tailor the products it runs to meet their company's specific needs.

We won't attempt to list all of the IBM licensed programs for z/OS (hundreds exist); some common choices include:

Security system
: z/OS provides a framework for customers to add security through the addition of a security management product (IBM's licensed program is Resource Access Control Facility or RACF®). Non-IBM security system licensed programs are also available.

Compilers
: z/OS includes an assembler and a C compiler. Other compilers, such as the COBOL compiler, and the PL/1 compiler are offered as separate products.

Relational database
: One example is DB2. Other types of database products, such as hierarchical databases, are also available.

Transaction processing facility
: IBM offers several, including:
  - Customer Information Control System (CICS)
  - Information Management System (IMS)
  - WebSphere® Application Server for z/OS

Sort program
: Fast, efficient sorting of large amounts of data is highly desirable in batch processing. IBM and other vendors offer sophisticated sorting products.

A large variety of utility programs
: For example, the System Display and Search Facility (SDSF) program that is used to view output from batch jobs is a licensed program. Not every installation purchases SDSF; alternative products are available.

Middleware for z/OS

Middleware is typically something between the operating system and an end user or end-user applications. It supplies major functions not provided by the operating system.

As commonly used, the term usually applies to major software products such as database managers, transaction monitors, Web servers, and so forth. **Subsystem** is another term often used for this type of software. These software products are usually licensed programs, although there are notable exceptions, such as the HTTP Server.

z/OS is a base for using many middleware products and functions. It is commonplace to run a variety of diverse middleware functions, with multiple instances of some. The routine use of wide-ranging workloads (mixtures of batch, transactions, Web serving, database queries and updates, and so on) is characteristic of z/OS.

Typical z/OS middleware includes:
- Database systems
- Web servers
- Message queueing and routing functions
- Transaction managers
- Java™ virtual machines
- Extensible Markup Language (XML) processing functions

A middleware product often includes an application programming interface (API). In some cases, applications are written to run completely under the control of this middleware API, while in other cases it is used only for unique purposes. Some examples of mainframe middleware APIs include:
- The WebSphere suite of products, which provides a complete API that is portable across multiple operating systems. Among these products, WebSphere MQ provides cross-platform APIs and inter-platform messaging.
- The DB2 database management product, which provides an API (expressed in the SQL language) that is used with many different languages and applications.

A Web server is considered to be middleware and Web programming (Web pages, CGIs, and so forth) is largely coded to the interfaces and standards presented by the Web server instead of the interfaces presented by the operating system. Java is another example in which applications are written to run under a Java Virtual Machine (JVM) and are largely independent of the operating system being used.

A brief comparison of z/OS and UNIX

What would we find if we compared z/OS and UNIX? In many cases, we'd find that quite a few concepts would be mutually understandable to users of either operating system, despite the differences in terminology.

For experienced UNIX users, Mapping UNIX to z/OS terms and concepts provides a small sampling of familiar computing terms and concepts. As a new user of z/OS, many of the z/OS terms will sound unfamiliar to you. As you work through this information center, however, the z/OS meanings will be explained and you will find that many elements of UNIX have analogs in z/OS.

A major difference for UNIX users moving to z/OS is the idea that the user is just one of many other users. In moving from a UNIX system to the z/OS environment, users typically ask questions such as ″Can I have the root password because I need to do...″ or ″Would you change this or that and restart the system?″ It is important for new z/OS users to understand that potentially thousands of other users are active on the same system, and so the scope of user actions and system restarts in z/OS and z/OS UNIX are carefully controlled to avoid negatively affecting other users and applications.

Under z/OS, there does not exist a single root password or root user. User IDs are external to z/OS UNIX System Services. User IDs are maintained in a security database that is shared with both UNIX and non-UNIX functions in the z/OS system, and possibly even shared with other z/OS systems. Typically, some user IDs have root authority, but these remain individual user IDs with individual passwords. Also, some user IDs do not normally have root authority, but can switch to ″root″ when circumstances require it.

Both z/OS and UNIX provide APIs to allow in-memory data to be shared between processes. In z/OS, a user can access another user's address spaces directly through **cross-memory services**. Similarly, UNIX has the concept of Shared Memory functions, and these can be used on UNIX without special authority.

z/OS cross-memory services, however, require the issuing program to have special authority, controlled by the authorized program facility (APF). This method allows efficient and secure access to data owned by others, data owned by the user but stored in another address space for convenience, and for rapid and secure communication with services like transaction managers and database managers.

Table 1. Mapping UNIX to z/OS terms and concepts

Term or concept	**UNIX**	**z/OS**
Start the operating system	Boot the system	IPL (initial program load) the system.
Virtual storage given to each user of the system	Users get whatever virtual storage they need to reference, within the limits of the hardware and operating system.	Users each get an address space, a range of addresses extending to 2 GB (or even 16 EB) of virtual storage, though some of this storage contains system code that is common for all users.
Data storage	Files	Data sets (sometimes called files)
Data format	Byte orientation; organization of the data is provided by the application	Record orientation; often an 80-byte record, reflecting the traditional punched card image
System configuration data	The /etc file system controls characteristics.	Parameters in PARMLIB control how the system IPLs and how address spaces behave.
Scripting languages	Shell scripts, Perl, awk, and other languages	CLISTS (command lists) and REXX™ execs

Table 1. Mapping UNIX to z/OS terms and concepts (continued)

Term or concept	UNIX	z/OS
Smallest element that performs work	A thread. The kernel supports multiple threads.	A task or a service request block (SRB). The z/OS base control program (BCP) supports multiple tasks and SRBs.
A long-running unit of work	A daemon	A started task or a long-running job; often this is a subsystem of z/OS.
Order in which the system searches for programs to run	Programs are loaded from the file system according to the user's PATH environmental variable (a list of directories to be searched).	The system searches the following libraries for the program to be loaded: TASKLIB, STEPLIB, JOBLIB, LPALST, and the linklist.
Using the system interactively	Users **log in** to systems and execute shell sessions in the shell environment. They can issue the **rlogin** or **telnet** commands to connect to the system. Each user can have many login sessions open at once.	Users **log on** to the system through TSO/E and its panel-driven interface, ISPF. A user ID is limited to having only one TSO/E logon session active at a time. Users can also log in to a z/OS UNIX shell environment using telnet, rlogin, or ssh.
Editing data or code	Many editors exist, such as vi, ed, sed, and emacs.	ISPF editor
Source and destination for input and output data	stdin and stdout	SYSIN and SYSOUT • SYSUT1 and SYSUT2 are used for utilities. • SYSTSIN and SYSTSPRT are used for TSO/E users.
Managing programs	The **ps** shell command allows users to view processes and threads, and kill jobs with the **kill** command.	SDSF allows users to view and terminate their jobs.

Chapter 2. z/OS storage constructs: File systems, data sets, and more

In working with the z/OS operating system, you must understand data sets, the files that contain programs and data. The characteristics of traditional z/OS data sets differ considerably from the file systems used in UNIX and PC systems. To make matters even more interesting, you can also create UNIX file systems on z/OS, with the common characteristics of UNIX systems.

z/OS manages data by means of **data sets**. The term data set refers to a file that contains one or more records. A data set can be a source program, a library of programs, or a file of data records used by a processing program. Data set **records** are the basic unit of information used by a processing program.

Users must define the amount of space to be allocated for a data set (before it is used), or these allocations must be automated through the use of DFSMS. With DFSMS, the z/OS system programmer or storage administrator can define performance goals and data availability requirements, create model data definitions for typical data sets, and automate data backup. DFSMS can automatically assign, based on installation policy, those services and data definition attributes to data sets when they are created. Other storage management-related products can be used to determine data placement, manage data backup, control space usage, and provide data security.

Almost all z/OS data processing is record-oriented. Byte-stream files are not present in traditional processing, although they are a standard part of z/OS UNIX. z/OS records and physical blocks follow one of several well-defined formats. Most data sets have DCB attributes that include the record format (RECFM), the maximum logical record length (LRECL), and the maximum block size (BLKSIZE).

z/OS libraries are known as partitioned data sets (PDS or PDSE) and contain members. Source programs, system and application control parameters, JCL, and executable modules are almost always contained in libraries.

Virtual storage access method (VSAM) is an access method that provides much more complex functions than other disk access methods. VSAM is primarily for applications and cannot be edited with ISPF.

z/OS data sets have names with a maximum of 44 uppercase characters, divided by periods into qualifiers with a maximum of 8 bytes per qualifier name. The high-level qualifier (HLQ) may be fixed by system security controls, but the rest of a data set name is assigned by the user. A number of conventions exist for these names.

An existing data set can be located when the data set name, volume, and device type are known. These requirements can be shortened to knowing only the data set name if the data set is cataloged. The system catalog is a single logical function, although its data may be spread across the master catalog and many user catalogs. In practice, almost all disk data sets are cataloged. One side effect of this is that all (cataloged) data sets must have unique names.

© Copyright IBM Corp. 2006, 2008

A file in the UNIX file system can be either a text file or a binary file. In a text file each line of text is separated by a newline delimiter. A binary file consists of sequences of binary words (byte stream), and no record concept other than the structure defined by an application exists. An application reading the file is responsible for interpreting the format of the data. z/OS treats an entire UNIX file system hierarchy as a collection of data sets. Each data set is a mountable file system.

What is a data set?

z/OS manages data by means of data sets. The term **data set** refers to a file that contains one or more **records**. The record is the basic unit of information used by a program running on z/OS.

Any named group of records is called a data set. Data sets can hold information such as medical records or insurance records, to be used by a program running on the system. Data sets are also used to store information needed by applications or the operating system itself, such as source programs, macro libraries, or system variables or parameters. For data sets that contain readable text, you can print them or display them on a console (many data sets contain load modules or other binary data that is not really printable). Data sets can be **cataloged**, which permits the data set to be referred to by name without specifying where it is stored.

In simplest terms, a **record** is a fixed number of bytes containing data. Often, a record collects related information that is treated as a unit, such as one item in a database or personnel data about one member of a department. The term **field** refers to a specific portion of a record used for a particular category of data, such as an employee's name or department.

The records in a data set can be organized in various ways, depending on how we plan to access the information. If you write an application program that processes things like personnel data, for example, your program can define a record format for each person's data.

There are many different types of data sets in z/OS, and different methods for accessing them. Among the most commonly used types are:

Sequential
: In a **sequential data set,** records are data items that are stored consecutively. To retrieve the tenth item in the data set, for example, the system must first pass the preceding nine items. Data items that must all be used in sequence, like the alphabetical list of names in a classroom roster, are best stored in a sequential data set.

Partitioned
: A partitioned data set or PDS consists of a **directory** and **members**. The directory holds the address of each member and thus makes it possible for programs or the operating system to access each member directly. Each member, however, consists of sequentially stored records. Partitioned data sets are often called **libraries**. Programs are stored as members of partitioned data sets. Generally, the operating system loads the members of a PDS into storage sequentially, but it can access members directly when selecting a program for execution.

VSAM
: In a Virtual Storage Access Method (VSAM) key sequenced data set (KSDS), records are data items that are stored with control information

(keys) so that the system can retrieve an item without searching all preceding items in the data set. VSAM KSDS data sets are ideal for data items that are used frequently and in an unpredictable order.

Quick reference: Data set structure

Working with data sets requires an understanding of the physical and logical structure of a data set, and how z/OS accesses information in the data set.

Data set
In z/OS, a *data set* is a named collection of related data records that is stored and retrieved by an assigned name. A data set is equivalent to a file in other operating systems. Data sets are stored on tape or disks.

Direct Access Storage Device (DASD)
DASD is another name for a disk drive. Additional synonyms include: disk volume, disk pack, or Head Disk Assembly (HDA).

Space
Disk space is allocated in units called cylinders, tracks, or blocks.

Cylinder
A disk drive contains *cylinders*. A cylinder is a unit of storage on a count-key-data (CKD) device with a fixed number of tracks.

Track
Cylinders contain *tracks*, which are circular paths on the surface of a disk or diskette on which information is magnetically recorded and from which recorded information is read. Tracks are in count-key-data (CKD) format, which means that each track contains fields that indicate the start of the track and the space used, followed by records containing three fields:

- The count field defines the length of the record
- The key field contains optional accounting information
- The data field contains the user data

Record
Tracks contain records. A *record* is some number of bytes containing data. The record is the basic unit of information used by a program running on z/OS.

- Records have a logical record length (abbreviated as LRECL); different types of DASD impose different maximum lengths for records.
- Records are either fixed length or variable length in a given data set. Traditional z/OS data sets have one of five record formats (abbreviated as RECFM): Fixed (F), fixed blocked (FB), variable (V), variable blocked (VB), or undefined (U).

Blocks
Records can be grouped into data *blocks*, which are the units of recording on disk. Blocking makes processing more efficient because z/OS can access an entire block at once instead of reading or writing records individually.

Block size (abbreviated as BLKSIZE) is the physical block size written on the disk for fixed (F) and fixed block (FB) records. For variable and undefined (V, VB, and U) records, block size is the maximum physical block size that can be used for the data set.

Extents
Space for a disk data set is assigned in primary and secondary *extents*. An extent is a contiguous number of disk drive tracks, cylinders, or blocks. Data sets can increase in extents as they grow. As with blocking, the use of extents is more efficient because reading or writing contiguous tracks is faster than reading or writing data that is scattered over the disk.

Volume
The term *volume* is often used to refer to a disk.

Volume serial
The six-character name of a disk or tape volume, such as TEST01.

Device type
A model or type of disk device, such as 3390.

Organization
The method of processing a data set, such as sequential.

Where are data sets stored?

z/OS 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, although some 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.

The term **DASD** applies to disks or simulated equivalents of disks. All types of data sets can be stored on DASD (only sequential data sets can be stored on magnetic tape). You use DASD volumes for storing data and executable programs, including the operating system itself, and for temporary working storage. You can use one DASD volume for many different data sets, and reallocate or reuse space on the volume.

To enable the system to locate a specific data set quickly, z/OS includes 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. z/OS requires that the master catalog reside on a DASD that is always mounted on a drive that is online to the system.

What are access methods?

An access method defines the technique that is used to store and retrieve data. Access methods have their own data set structures to organize data, system-provided programs (or **macros**) to define data sets, and utility programs to process data sets.

Access methods are identified primarily by the data set organization. z/OS users, for example, use the basic sequential access method (BSAM) or queued sequential access method (QSAM) with sequential data sets.

There are times when an access method identified with one organization can be used to process a data set organized in a different manner. For example, a sequential data set (not extended-format data set) created using BSAM can be processed by the basic direct access method (BDAM), and vice versa. Another example is UNIX files, which you can process using BSAM, QSAM, basic partitioned access method (BPAM), or virtual storage access method (VSAM).

Commonly used access methods include the following:

QSAM
QSAM (Queued Sequential Access Method) is a heavily used access method. QSAM arranges records sequentially in the order that they are entered to form sequential data sets, and anticipates the need for records

based on their order. The system organizes records with other records. To improve performance, QSAM reads these records into storage before they are requested, a technique known as queued access.

BSAM
BSAM (Basic Sequential Access Method) is used for special cases. BSAM arranges records sequentially in the order in which they are entered. Unlike QSAM, however, the user– rather than the system– organizes records with other records into blocks.

BDAM
BDAM (Basic Direct Access Method), which is becoming obsolete, arranges records in any sequence your program indicates, and retrieves records by actual or relative address. If you do not know the exact location of a record, you can specify a point in the data set where a search for the record is to begin. Data sets organized this way are called direct data sets.

BPAM BPAM (Basic Partitioned Access Method) arranges records as members of a partitioned data set (PDS) or a partitioned data set extended (PDSE) on DASD. You can use BPAM to view a UNIX directory and its files as if it were a PDS. (You can view each PDS, PDSE, or UNIX member sequentially with BSAM or QSAM.)

VSAM
VSAM (Virtual Sequential Access Method) is used for more complex applications. VSAM arranges records by an index key, relative record number, or relative byte addressing. VSAM is used for direct or sequential processing of fixed-length and variable-length records on DASD. Data that is organized by VSAM is cataloged for easy retrieval.

What are DASD volumes and labels?

DASD volumes are used for storing data and executable programs (including the operating system itself), and for temporary working storage. DASD labels identify DASD volumes and the data sets they contain.

One DASD volume can be used for many different data sets, and space on it can be reallocated and reused. On a volume, the name of a data set must be unique. A data set can be located by device type, volume serial number, and data set name. This structure is unlike the file tree of a UNIX system. The basic z/OS file structure is not hierarchical. z/OS data sets have no equivalent to a path name.

Although DASD volumes differ in physical appearance, capacity, and speed, they are similar in data recording, data checking, data format, and programming. The recording surface of each volume is divided into many concentric **tracks**. The number of tracks and their capacity vary with the device. Each device has an access mechanism that contains read/write heads to transfer data as the recording surface rotates past them.

The operating system uses groups of labels to identify DASD volumes and the data sets they contain. Customer application programs generally do not use these labels directly. DASD volumes must use standard labels. Standard labels include a volume label, a data set label for each data set, and optional user labels. A volume label, stored at track 0 of cylinder 0, identifies each DASD volume.

The z/OS system programmer or storage administrator uses the ICKDSF utility program to initialize each DASD volume before it is used on the system. ICKDSF generates the volume label and builds the volume table of contents (VTOC), a

structure that contains the data set labels. The system programmer can also use ICKDSF to scan a volume to ensure that it is usable and to reformat all the tracks.

Allocating a data set

To use a data set, you first allocate it (establish a link to it), then access the data using macros for the access method that you have chosen.

The **allocation** of a data set means either or both of two things:

- To set aside (create) space for a new data set on a disk.
- To establish a logical link between a job step and any data set.

You can allocate a data set using ISPF panel option 3.2. Other ways to allocate a data set include the following methods:

Access method services
: You can allocate data sets through a multifunction services program called access method services. Access method services include commonly used commands for working with data sets, as ALLOCATE, ALTER, DELETE, and PRINT.

ALLOCATE
: You can use the TSO ALLOCATE command to create data sets. The command actually guides you through the allocation values that you must specify.

ISPF menus
: One menu guides the user through allocation of a data set.

JCL
: You can use specific job control language (JCL) statements to allocate data sets.

How are data sets named?

When you allocate a new data set, you must give the data set a unique name.

A data set name can be one name segment, or a series of joined name segments. Each name segment represents a level of qualification. For example, the data set name VERA.LUZ.DATA is composed of three name segments. The first name on the left is called the high-level qualifier (HLQ), the last name on the right is the lowest-level qualifier (LLQ).

Segments or **qualifiers** are limited to eight characters, the first of which must be alphabetic (A to Z) or special (# @ $). The remaining seven characters are either alphabetic, numeric (0 - 9), special, a hyphen (-). Name segments are separated by a period (.).

Including all name segments and periods, the length of the data set name must not exceed 44 characters. Thus, a maximum of 22 name segments can make up a data set name.

For example, the following names are not valid data set names:

- Name with a qualifier that is longer than eight characters (HLQ.ABCDEFGHI.XYZ)
- Name containing two successive periods (HLQ..ABC)
- Name that ends with a period (HLQ.ABC.)
- Name that contains a qualifier that does not start with an alphabetic or special character (HLQ.123.XYZ)

The HLQ for a user's data sets is typically controlled by the security system. There are a number of conventions for the remainder of the name. These are **conventions**, not rules, but are widely used. They include the following:

- The letters LIB somewhere in the name indicate that the data set is a library. The letters PDS are a lesser-used alternative for this convention.
- The letters CNTL, JCL, or JOB somewhere in the name typically indicate the data set contains JCL (but might not be exclusively devoted to JCL).
- The letters LOAD, LOADLIB, or LINKLIB in the name indicate that the data set contains executables. (A library with z/OS executable modules must be devoted solely to executable modules.)
- The letters PROC, PRC, or PROCLIB indicate a library of JCL procedures.
- Various combinations are used to indicate source code for a specific language, for example COBOL, Assembler, FORTRAN, PL/I, JAVA, C, or C++.
- A portion of a data set name may indicate a specific project, such as PAYROLL.
- Using too many qualifiers is considered poor practice. For example, the following name is a valid data set name (upper case, does not exceed 44 bytes, no special characters) but it is not very meaningful.

  ```
  P390A.A.B.C.D.E.F.G.H.I.J.K.L.M.N.O.P.Q.R.S
  ```

  A good practice is for a data set name to contain three or four qualifiers.

How is space allocated on DASD volumes?

To allocate a data set using JCL, you specify the amount of space required in blocks, records, tracks, or cylinders.

When creating a DASD data set, you specify the amount of space needed explicitly through the SPACE parameter, or implicitly by using the information available in a data class. The system can use a data class if SMS is active even if the data set is not SMS-managed. For system-managed data sets, the system selects the volumes, saving you from having to specify a volume when you allocate a data set.

If you specify your space request by average record length, space allocation is independent of device type. Device independence is especially important to system-managed storage. A logical record length (LRECL) is a unit of information about a unit of processing (for example, a customer, an account, a payroll employee, and so on). It is the smallest amount of data to be processed, and it is comprised of fields that contain information recognized by the processing application. The maximum length of a logical record (LRECL) is limited by the physical size of the used media.

Logical records, when located on DASD, tape, or optical devices, are grouped within physical records named blocks. BLKSIZE indicates the length of those blocks. Each block of data on a DASD volume has a distinct location and a unique address, thus making it possible to find any block without extensive searching. Logical records can be stored and retrieved either directly or sequentially.

When the amount of space required is expressed in blocks, you must specify the number and average length of the blocks within the data set.

Let us take an example of a request for disk storage as follows:

- Average block length in bytes = 300
- Primary quantity (number) of blocks = 5000
- Secondary quantity of blocks, to be allocated if the primary quantity gets filled with data = 100

From this information, the operating system estimates and allocates the amount of disk space required.

Space for a disk data set is assigned in extents. An extent is a **contiguous** number of disk drive tracks, cylinders, or blocks. Data sets can increase in extents as they grow. Older types of data sets can have up to 16 extents per volume. Newer types of data sets can have up to 128 extents per volume or 255 extents total on multiple volumes.

Extents are relevant when you are not using PDSEs and have to manage the space yourself, rather than through DFSMS. Here, you want the data set to fit into a single extent to maximize disk performance. Reading or writing contiguous tracks is faster than reading or writing tracks scattered over the disk, as might be the case if tracks were allocated dynamically. But if there is not sufficient contiguous space, a data set goes into extents.

Data set record formats

Traditional z/OS data sets are **record oriented**, and have one of five possible formats.

In normal usage, there are no byte stream files such as are found in PC and UNIX systems. (z/OS UNIX has byte stream files, and byte stream functions exist in other specialized areas. These are not considered to be traditional data sets.)

In z/OS, there are no new line (NL) or carriage return and line feed (CR+LF) characters to denote the end of a record. Records are either fixed length or variable length in a given data set. When editing a data set with ISPF, for example, each line is a record.

Traditional z/OS data sets have one of five record formats, as follows:

F (Fixed)
: Fixed means that one physical block on disk is one logical record and all the blocks and records are the same size. This format is seldom used.

FB (Fixed Blocked)
: This format designation means that several logical records are combined into one physical block. This format can provide efficient space utilization and operation. This format is commonly used for fixed-length records.

V (Variable)
: This format has one logical record as one physical block. A variable-length logical record consists of a record descriptor word (RDW) followed by the data. The record descriptor word is a 4-byte field describing the record. The first 2 bytes contain the length of the logical record (including the 4-byte RDW). The length can be from 4 to 32,760 bytes. All bits of the third and fourth bytes must be 0, because other values are used for spanned records. This format is seldom used.

VB (Variable Blocked)
: This format places several variable-length logical records (each with an RDW) in one physical block. The software must place an additional Block Descriptor Word (BDW) at the beginning of the block, containing the total length of the block.

U (Undefined)
: This format consists of variable-length physical records and blocks with no predefined structure. Although this format may appear attractive for many unusual applications, it is normally used only for executable modules.

We must stress the difference between a block and a record: a block is what is written on disk, while a record is a logical entity.

The terminology here is pervasive throughout z/OS literature. The key terms are:

- Block Size (BLKSIZE) is the physical block size written on the disk for F and FB records. For V, VB, and U records, it is the maximum physical block size that can be used for the data set.
- Logical Record Size (LRECL) is the logical record size (for formats F and FB) or the maximum allowed logical record size (for formats V and VB) for the data set. Format U records have no LRECL.
- Record Format (RECFM) is F, FB, V, VB, or U as just described.

These terms are known as data control block (DCB) characteristics, named for the control block where they may be defined in an assembly language program. The user is often expected to specify these parameters when creating a new data set. The type and length of a data set are defined by its record format (RECFM) and logical record length (LRECL). Fixed-length data sets have a RECFM of F, FB, FBS, and so on. Variable-length data sets have a RECFM of V, VB, VBS, and so on.

A data set with RECFM=FB and LRECL=25 is a fixed-length (FB) data set with a record length of 25 bytes (the B is for blocked). For an FB data set, the LRECL tells you the length of each record in the data set; all of the records are the same length. The first data byte of an FB record is in position 1. A record in an FB data set with LRECL=25 might look like this:

```
Positions 1-3: Country Code = 'USA'
Positions 4-5: State Code = 'CA'
Positions 6-25: City = 'San Jose' padded with 12 blanks on the right
```

A data set with RECFM=VB and LRECL=25 is a variable-length (VB) data set with a maximum record length of 25 bytes. In a VB data set, the records can have different lengths. The first four bytes of each record contain the RDW, and the first two bytes of the RDW contain the length of that record (in binary). The first data byte of a VB record is in position 5, after the 4-byte RDW in positions 1-4. A record in a VB data set with LRECL=25 might look like this:

```
Positions 1-2: Length in RDW = hex 0011 = decimal 17
Positions 3-4: Zeros in RDW = hex 0000 = decimal 0
Positions 5-7: Country Code = 'USA'
Positions 8-9: State Code = 'CA'
Positions 10-17: City = 'San Jose'
```

Figure 11 on page 44 shows the relationship between records and blocks for each of the five record formats.

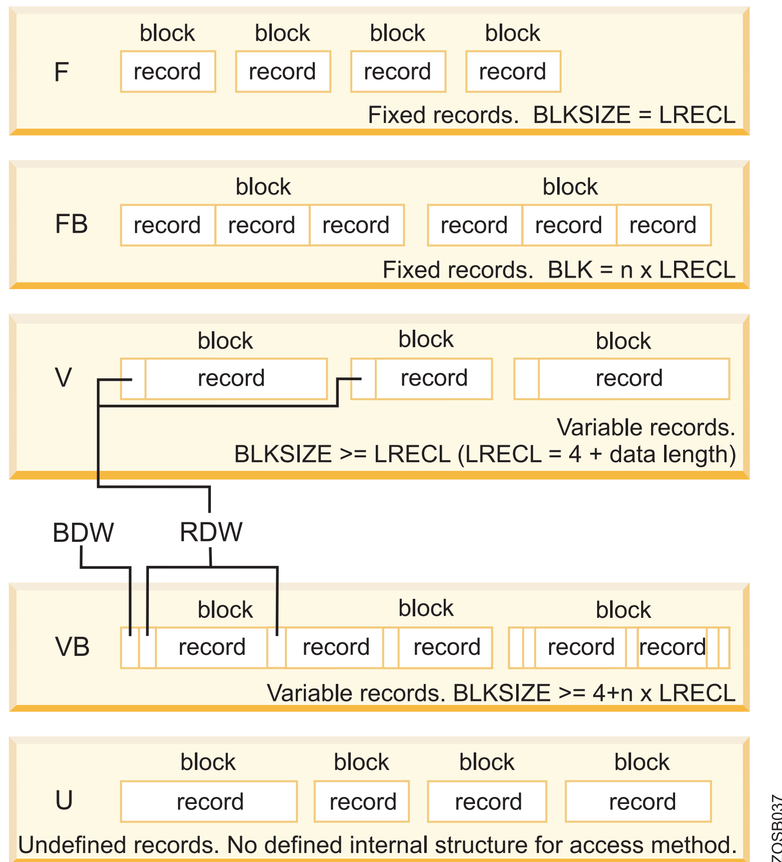


Figure 11. Basic record formats

Types of data sets

z/OS has many different types of data sets.

Data sets can be sequential or partitioned:

- In a *sequential* data set, records are data items that are stored consecutively.
- A *partitioned* 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. Each member consists of sequentially stored records.

Partitioned data sets are often called *libraries*. By convention, libraries often have the letters “LIB” in the data set name. Also by convention, programs and procedures are stored in separate libraries; within a library, each program or procedure is stored as a separate member of the partitioned data set.

Data sets can be permanent or temporary:

- Most permanent data sets exist before a job starts and persist after a job step completes. Some permanent data sets are created during a job step and persist after the job completes.
- Temporary data sets generally are used to pass data from one job step to another, and exist only during the life cycle of the job.

Data sets can be cataloged, which permits the data set to be referred to by name without specifying where the data set is stored. A catalog describes data set attributes and indicates the devices on which a data set is located. In z/OS, the master catalog and user catalogs store the locations of data sets.

Why is a PDS structured like that?

A PDS data set offers a simple and efficient way to organize related groups of sequential files. As for most things, the PDS structure offers advantages but also has some disadvantages.

The PDS structure was designed to provide efficient access to libraries of related members, whether they be load modules, program source modules, JCL or many other types of content. Many system data sets are also kept in PDS data sets, especially when they consist of many small, related files. For example, the definitions for ISPF panels are kept in PDS data sets.

A primary use of ISPF is to create and manipulate PDS data sets. These data sets typically consist of source code for programs, text for manuals or help screens, or JCL to allocate data sets and run programs.

A PDS has the following advantages for z/OS users:

- Grouping of related data sets under a single name makes z/OS data management easier. Files stored as members of a PDS can be processed either individually or all the members can be processed as a unit.
- Because the space allocated for z/OS data sets always starts at a track boundary on disk, using a PDS is a way to store more than one small data set on a track. This method saves you disk space if you have many data sets that are much smaller than a track. A track is 56,664 bytes for a 3390 disk device.
- Members of a PDS can be used as sequential data sets, and they can be appended (or **concatenated**) to sequential data sets.
- Multiple PDS data sets can be concatenated to form large libraries.
- PDS data sets are easy to create with JCL or ISPF; they are easy to manipulate with ISPF utilities or TSO commands.

PDS data sets are simple, flexible, and widely used. However, some aspects of the PDS design affect both performance and the efficient use of disk storage, as follows:

- Wasted space

  When a member in a PDS is replaced, the new data area is written to a new section within the storage allocated to the PDS. When a member is deleted, its pointer is deleted too, so there is no mechanism to reuse its space. This wasted space is often called **gas** and must be periodically removed by reorganizing the PDS, for example, by using the utility IEBCOPY to compress it.
- Limited directory size

  The size of a PDS directory is set at allocation time. As the data set grows, it can acquire more space in units of the amount you specified as its secondary space. These extra units are called **secondary extents**.

  However, you can store only a fixed number of member entries in the PDS directory because its size is fixed when the data set is allocated. If you need to store more entries than the existing directory space will hold, you have to allocate a new PDS with more directory blocks and copy the members from the old data set into it. This fact means that when you allocate a PDS, you must calculate the amount of directory space you need.

- Lengthy directory searches

  An entry in a PDS directory consists of a name and a pointer to the location of the member. Entries are stored in alphabetical order of the member names. Inserting an entry near the front of a large directory can cause a large amount of I/O activity, as all the entries behind the new one are moved along to make room for it.

  Also, entries are searched sequentially in alphabetical order. If the directory is very large and the members small, it might take longer to search the directory than to retrieve the member when its location is found.

What is a PDSE?

The acronym PDSE stands for **partitioned data set extended**. A PDSE consists of a directory and zero or more members, just like a PDS... But there are some differences between the two.

A PDSE can be created with JCL, TSO/E, and ISPF, just like a PDS, and can be processed with the same access methods. PDSE data sets are stored only on DASD, not on tape.

The directory can expand automatically as needed, up to the addressing limit of 522,236 members. It also has an index, which provides a fast search for member names. Space from deleted or moved members is automatically reused for new members, so you do not have to compress a PDSE to remove wasted space. Each member of a PDSE can have up to 15,728,639 records. A PDSE can have a maximum of 123 extents, but it cannot extend beyond one volume. When a directory of a PDSE is in use, it is kept in processor storage for fast access.

PDSE data sets can be used in place of nearly all PDS data sets that are used to store data. But the PDSE format is not intended as a PDS replacement. When a PDSE is used to store load modules, it stores them in structures called **program objects**.

In many ways, a PDSE is similar to a PDS. Each member name can be eight bytes long. For accessing a PDS directory or member, most PDSE interfaces are indistinguishable from PDS interfaces. PDS and PDSE data sets are processed using the same access methods (for example, BSAM, QSAM and BPAM). Within a given PDS or PDSE, the members must use the same access method.

However, PDSE data sets have a different internal format, which gives them increased usability. You can use a PDSE in place of a PDS to store data or programs. In a PDS, you store programs as **load modules**. In a PDSE, you store programs as program objects. If you want to store a load module in a PDSE, you must first convert it into a program object (using the IEBCOPY utility).

PDSE data sets have several features that can improve user productivity and system performance. The main advantage of using a PDSE over a PDS is that a PDSE automatically reuses space within the data set without the need for anyone to periodically run a utility to reorganize it. The system reclaims space automatically whenever a member is deleted or replaced, and returns it to the pool of space available for allocation to other members of the same PDSE. The space can be reused without having to do an IEBCOPY compress.

Also, the size of a PDS directory is fixed regardless of the number of members in it, while the size of a PDSE directory is flexible and expands to fit the members stored in it.

Other advantages of PDSE data sets follow:

- PDSE members can be shared. This characteristic makes it easier to maintain the integrity of the PDSE when modifying separate members of the PDSE at the same time.
- The system requires less time to search a PDSE directory. The PDSE directory, which is indexed, is searched using that index. The PDS directory, which is organized alphabetically, is searched sequentially. The system might cache in storage directories of frequently used PDSE data sets.
- You may create multiple PDSE members at the same time. For example, you can open two data control blocks (DCBs) to the same PDSE and write two members at the same time.
- PDSE data sets contain up to 123 extents. An extent is a continuous area of space on a DASD storage volume, occupied by or reserved for a specific data set.
- When written to DASD, logical records are extracted from the user's blocks and reblocked. When read, records in a PDSE are reblocked into the block size specified in the DCB. The block size used for the reblocking can differ from the original block size.

What happens when a data set runs out of space?

When you allocate a data set, you reserve a certain amount of space in units of blocks, tracks, or cylinders on a storage disk. If you use up that space, the system displays the message SYSTEM ABEND '0D37,' or possibly B37 or E37.

This situation is something you will have to deal with if it occurs. If you are in an edit session, you will not be able to exit the session until you resolve the problem.

Among the things you can do to resolve a space shortage abend are:

- If the data set is a PDS, you can compress it by doing the following:
  1. Split (PF 2) the screen and select UTILITIES (option 3).
  2. Select LIBRARIES (option 1) on the Utility Selection Menu.
  3. Specify the name of the data set and enter `C` on the option line.
  4. When the data set is compressed, you should see the message `COMPRESS SUCCESSFUL`.
  5. You can then swap (PF 9) to the edit session and save the new material.
- Allocate a larger data set and copy into it by doing the following:
  1. Split (PF 2) the screen and select UTILITIES (option 3), then DATASET (option 2) from the other side of the split.
  2. Specify the name of the data set that received the abend to display its characteristics.
  3. Allocate another data set with more space.
  4. Select MOVE/COPY (option 3) on the Utility Selection Menu to copy members from the old data set to the new larger data set.
  5. Browse (option 1) the new data set to make sure everything was copied correctly.
  6. Swap (PF 9) back to the abending edit session, enter `CC` on the top line of input and the bottom line of input, enter `CREATE` on the command line, and press the Enter key.
  7. Enter the new, larger data set name and a member name to receive the copied information.

8. You again see the abending edit session. Enter CANCEL on the command line. Press the RETURN key (PF 4) key to exit the edit session.
9. Select DATASET (option 2) from the Utility Selection Menu to delete the old data set.
10. Rename the new data set to the old name.

- Cancel the new material entered in the edit session by entering CANCEL on the command line. You should then be able to exit without abending; however, all information that was not previously saved is lost.

What is VSAM?

The term **Virtual Storage Access Method** (VSAM) applies to both a data set type and the access method used to manage various user data types.

As an access method, VSAM provides much more complex functions than other disk access methods. VSAM keeps disk records in a unique format that is not understandable by other access methods.

VSAM is used primarily for applications. It is not used for source programs, JCL, or executable modules. VSAM files cannot be routinely displayed or edited with ISPF.

You can use VSAM to organize records into four types of data sets: key-sequenced, entry-sequenced, linear, or relative record. The primary difference among these types of data sets is the way their records are stored and accessed.

VSAM data sets are briefly described as follows:

Key Sequence Data Set (KSDS)
This type is the most common use for VSAM. Each record has one or more key fields and a record can be retrieved (or inserted) by key value. This provides random access to data. Records are of variable length.

Entry Sequence Data Set (ESDS)
This form of VSAM keeps records in sequential order. Records can be accessed sequentially. It is used by IMS, DB2, and z/OS UNIX.

Relative Record Data Set (RRDS)
This VSAM format allows retrieval of records by number; record 1, record 2, and so forth. This provides random access and assumes the application program has a way to derive the desired record numbers.

Linear Data Set (LDS)
This type is, in effect, a byte-stream data set and is the only form of a byte-stream data set in traditional z/OS files (as opposed to z/OS UNIX files). A number of z/OS system functions use this format heavily, but it is rarely used by application programs.

Several additional methods of accessing data in VSAM are not listed here. Most applications use VSAM for keyed data.

VSAM works with a logical data area known as a control interval (CI) that is diagrammed in Figure 12 on page 49. The default CI size is 4K bytes, but it can be up to 32K bytes. The CI contains data records, unused space, record descriptor fields (RDFs), and a CI descriptor field.

Figure 12. Simple VSAM control interval

Multiple CIs are placed in a control area (CA). A VSAM data set consists of control areas and index records. One form of index record is the sequence set, which is the lowest-level index pointing to a control interval.

VSAM data is always variable-length and records are automatically blocked in control intervals. The RECFM attributes (F, FB, V, VB, U) do not apply to VSAM, nor does the BLKSIZE attribute. You can use the Access Method Services (AMS) utility to define and delete VSAM structures, such as files and indexes. Defining a VSAM KSDS using AMS shows an example.

```
DEFINE CLUSTER -
(NAME(VWX.MYDATA) -
VOLUMES(VSER02) -
RECORDS(1000 500)) -
DATA -
(NAME(VWX.KSDATA) -
 KEYS(15 0) -
RECORDSIZE(250 250) -
BUFFERSPACE(25000) ) -
INDEX -
(NAME(VWX.KSINDEX) -
CATALOG (UCAT1)
```

Many details of VSAM processing are not included in this brief description. Most processing is handled transparently by VSAM; the application program merely retrieves, updates, deletes or adds records based on key values.

What is a VTOC?

z/OS uses a catalog and a volume table of contents (VTOC) on each DASD to manage the storage and placement of data sets.

z/OS requires a particular format for disks, which is shown in Figure 13 on page 50. Record 1 on the first track of the first cylinder provides the label for the disk. It contains the 6-character volume serial (volser) number and a pointer to the **volume table of contents** (VTOC), which can be located anywhere on the disk.

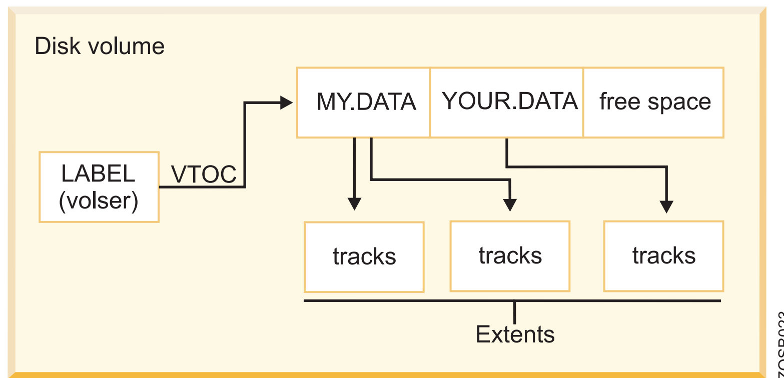


Figure 13. Disk label, VTOC, and extents

The VTOC lists the data sets that reside on its volume, along with information about the location and size of each data set, and other data set attributes. A standard z/OS utility program, ICKDSF, is used to create the label and VTOC. When a disk volume is initialized with ICKDSF, the owner can specify the location and size of the VTOC. The size can be quite variable, ranging from a few tracks to perhaps 100 tracks, depending on the expected use of the volume. More data sets on the disk volume require more space in the VTOC.

The VTOC also has entries for all the free space on the volume. Allocating space for a data set causes system routines to examine the free space records, update them, and create a new VTOC entry. Data sets are always an integral number of tracks (or cylinders) and start at the beginning of a track (or cylinder).

You can also create a VTOC with an index. The VTOC index is actually a data set with the name SYS1.VTOCIX.volser, which has entries arranged alphabetically by data set name with pointers to the VTOC entries. It also has bitmaps of the free space on the volume. A VTOC index allows the user to find the data set much faster.

What is a catalog?

A catalog describes data set attributes and indicates the volumes on which a data set is located.

When a data set is cataloged, it can be referred to by name without the user needing to specify where the data set is stored. Data sets can be cataloged, uncataloged, or recataloged. All system-managed DASD data sets are cataloged automatically in a catalog. Cataloging of data sets on magnetic tape is not required, but doing so can simplify users' jobs.

In z/OS, the master catalog and user catalogs store the locations of data sets. Both disk and tape data sets can be cataloged.

To find a data set that you have requested, z/OS must know three pieces of information:
- Data set name
- Volume name
- Unit (the volume device type, such as a 3390 disk or 3590 tape)

You can specify all three values on ISPF panels or in JCL. However, the unit device type and the volume are often not relevant to an end user or application program.

A system catalog is used to store and retrieve UNIT and VOLUME location of a data set. In its most basic form, a catalog can provide the unit device type and volume name for any data set that is cataloged. A system catalog provides a simple look-up function. With this facility the user need only provide a data set name.

A z/OS system always has at least one **master catalog**. If it has a single catalog, this catalog would be the master catalog and the location entries for all data sets would be stored in it. A single catalog, however, would be neither efficient nor flexible, so a typical z/OS system uses a master catalog and numerous user catalogs connected to it as shown in Figure 14 on page 52.

A **user catalog** stores the name and location of a data set (dsn/volume/unit). The master catalog usually stores only a data set high-level qualifier (HLQ) with the name of the user catalog, which contains the location of all data sets prefixed by this HLQ. The HLQ is called an alias.

In Figure 14 on page 52, **SYSTEM.MASTER.CATALOG** is the data set name of the master catalog. This master catalog stores the full data set name and location of all data sets with a SYS1 prefix such as SYS1.A1. Two HLQ (alias) entries were defined to the master catalog, IBMUSER and USER. The statement that defined IBMUSER included the data set name of the user catalog containing all the fully qualified IBMUSER data sets with their respective location. The same is true for USER HLQ (alias).

When SYS1.A1 is requested, the master catalog returns the location information, volume(WRK001) and unit(3390), to the requestor. When IBMUSER.A1 is requested, the master catalog redirects the request to USERCAT.IBM, then USERCAT.IBM returns the location information to the requestor.

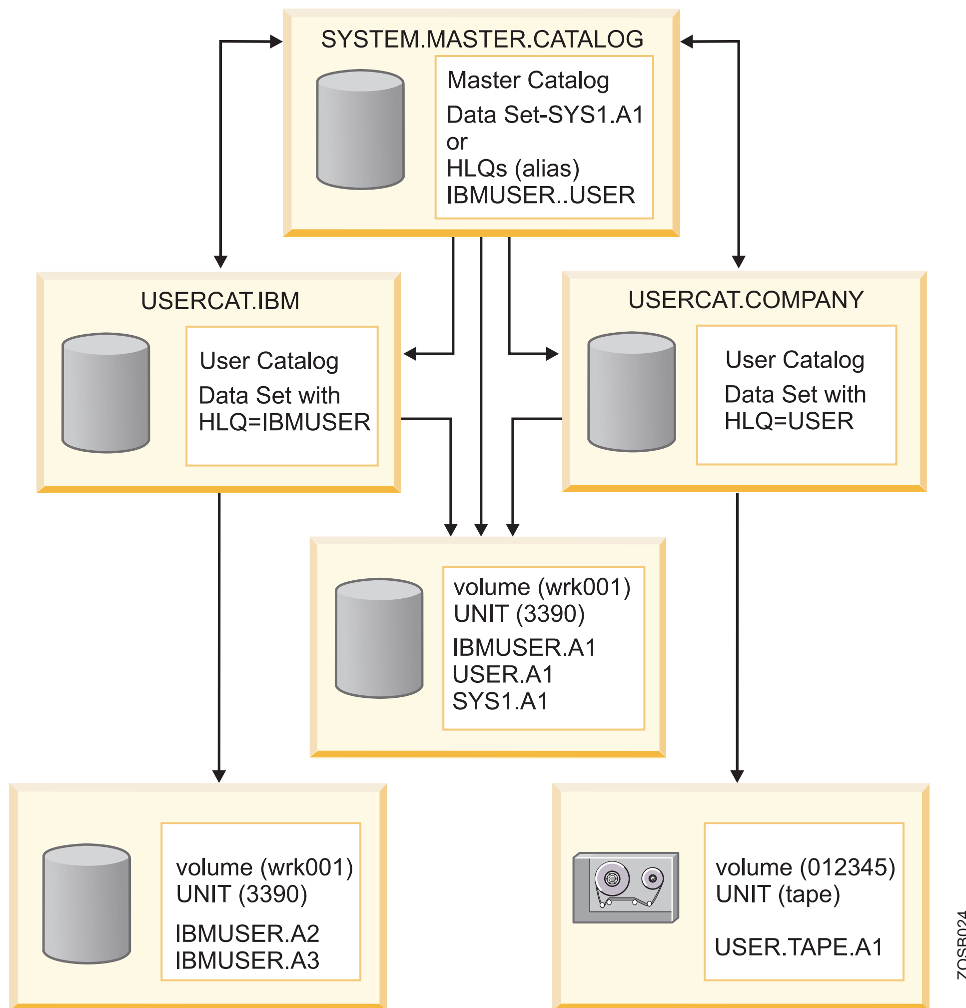


Figure 14. Catalog concept

Take, as a further example, the following DEFINE statements:

```
DEFINE ALIAS ( NAME ( IBMUSER ) RELATE ( USERCAT.IBM ) )
DEFINE ALIAS ( NAME ( USER ) RELATE ( USERCAT.COMPANY ) )
```

These statements are used to place IBMUSER and USER alias names in the master catalog with the name of the user catalog that will store the fully qualified data set names and location information. If IBMUSER.A1 is cataloged, a JCL statement to allocate it to the job would be:

```
//INPUT DD DSN=IBMUSER.A1,DISP=SHR
```

If IBMUSER.A1 is not cataloged, a JCL statement to allocate it to the job would be:

```
//INPUT DD DSN=IBMUSER.A1,DISP=SHR,VOL=SER=WRK001,UNIT=3390
```

As a general rule, all user data sets in a z/OS installation are cataloged. Uncataloged data sets are rarely needed and their use is often related to recovery problems or installation of new software. Data sets created through ISPF are automatically cataloged.

So, what happens if an installation loses its master catalog, or the master catalog somehow becomes corrupted? Such an occurrence would pose a serious problem and require swift recovery actions. To save this potential headache, most system

programmers define a back-up for the master catalog. The system programmer specifies this alternate master catalog during system start-up. In this case, it's recommended that the system programmer keep the alternate on a volume separate from that of the master catalog (to protect against a situation in which the volume becomes unavailable).

What is a generation data group?

In z/OS, it is possible to catalog successive updates or generations of related data, which are called generation data groups (GDGs).

Each data set within a GDG is called a generation or generation data set (GDS). A generation data group (GDG) is a collection of historically related non-VSAM data sets that are arranged in chronological order. That is, each data set is historically related to the others in the group.

Within a GDG, the generations can have like or unlike DCB attributes and data set organizations. If the attributes and organizations of all generations in a group are identical, the generations can be retrieved together as a single data set.

Advantages to grouping related data sets include:

- All of the data sets in the group can be referred to by a common name.
- The operating system is able to keep the generations in chronological order.
- Outdated or obsolete generations can be automatically deleted by the operating system.

Generation data sets have sequentially ordered absolute and relative names that represent their age. The operating system's catalog management routines use the absolute generation name. Older data sets have smaller absolute numbers. The relative name is a signed integer used to refer to the latest (0), the next to the latest (-1), and so forth, generation.

For example, the data set name LAB.PAYROLL(0) refers to the most recent data set of the group; LAB.PAYROLL(-1) refers to the second most recent data set; and so forth. The relative number can also be used to catalog a new generation (+1). A generation data group (GDG) base is allocated in a catalog before the generation data sets are cataloged. Each GDG is represented by a GDG base entry.

For new non-system-managed data sets, if you do not specify a volume and the data set is not opened, the system does not catalog the data set. New system-managed data sets are always cataloged when allocated, with the volume assigned from a storage group.

Role of DFSMS in managing space

The primary means of managing space in z/OS is through the Data Facility Storage Management Subsystem (DFSMS), which comprises a suite of related data and storage management products. DFSMS performs the essential data, storage, program, and device management functions of the system.

In a z/OS system, space management involves the allocation, placement, monitoring, migration, backup, recall, recovery, and deletion of data sets. These activities can be done either manually or through the use of automated processes. When data management is automated, the operating system determines object

placement and automatically manages data set backup, movement, space, and security. A typical z/OS production system includes both manual and automated processes for managing data sets.

Depending on how a z/OS system and its storage devices are configured, a user or program can directly control many aspects of data set usage, and in the early days of the operating system, users were required to do so. Increasingly, however, z/OS customers rely on installation-specified settings for data and resource management, and space management products, such as DFSMS, to automate the use of storage for data sets.

Data management includes these main tasks:
- Setting aside (allocating) space on DASD volumes.
- Automatically retrieving cataloged data sets by name.
- Mounting magnetic tape volumes in the drive.
- Establishing a logical connection between the application program and the medium.
- Controlling access to data.
- Transferring data between the application program and the medium.

DFSMS, together with hardware products and installation-specific settings for data and resource management, provides system-managed storage in a z/OS environment.

The heart of DFSMS is the Storage Management Subsystem (SMS). Using SMS, the system programmer or storage administrator defines policies that automate the management of storage and hardware devices. These policies describe data allocation characteristics, performance and availability goals, backup and retention requirements, and storage requirements for the system. SMS governs these policies for the system, and the Interactive Storage Management Facility (ISMF) provides the user interface for defining and maintaining the policies.

The data sets allocated through SMS are called system-managed data sets or SMS-managed data sets. When you allocate or define a data set to use SMS, you specify the data set requirements through a data class, a storage class, and a management class. Typically, you do not need to specify these classes because a storage administrator has set up automatic class selection (ACS) routines to determine which classes are used for a given data set.

DFSMS provides a set of constructs, user interfaces, and routines (using the DFSMS products) to help the storage administrator. The core logic of DFSMS, such as the ACS routines, ISMF code, and constructs, resides in DFSMSdfp™. DFSMShsm™ and DFSMSdss™ are involved in the management class construct. With DFSMS, the z/OS system programmer or storage administrator can define performance goals and data availability requirements, create model data definitions for typical data sets, and automate data backup. DFSMS can automatically assign, based on installation policy, those services and data definition attributes to data sets when they are created. IBM storage management-related products determine data placement, manage data backup, control space usage, and provide data security.

z/OS UNIX file systems

z/OS UNIX System Services (z/OS UNIX) allows z/OS users to create UNIX file systems and file system directory trees on z/OS, and to access UNIX files on z/OS and other systems.

Think of a UNIX file system as a container that holds part of the entire UNIX directory tree. Unlike a traditional z/OS library, a UNIX file system is hierarchical and byte-oriented. To find a file in a UNIX file system, you search one or more directories (see Figure 15). There is no concept of a z/OS catalog that points directly to a file.

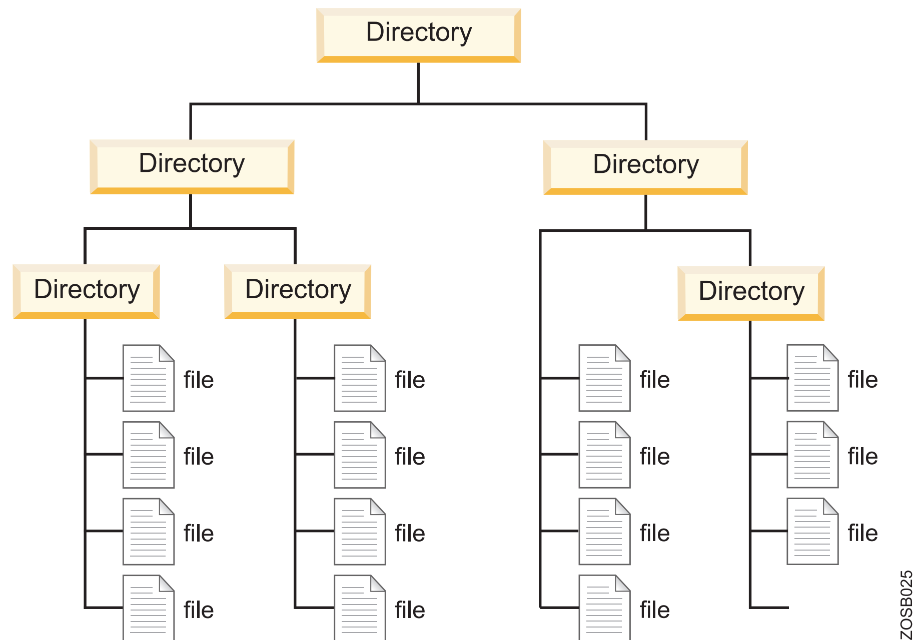


Figure 15. A hierarchical file system structure

In z/OS, a UNIX file system is mounted over an empty directory by the system programmer (or a user with mount authority).

You can use the following file system types with z/OS UNIX:

- zSeries File System (zFS), which is a file system that stores files in VSAM linear data sets.
- Hierarchical file system (HFS), a mountable file system, which is being phased out by zFS.
- z/OS Network File System (z/OS NFS), which allows a z/OS system to access a remote UNIX (z/OS or non-z/OS) file system over TCP/IP, as if it were part of the local z/OS directory tree.
- Temporary file system (TFS), which is a temporary, in-memory physical file system that supports in-storage mountable file systems.

As with other UNIX file systems, a path name identifies a file and consists of directory names and a file name. A fully qualified file name, which consists of the name of each directory in the path to a file plus the file name itself, can be up to 1023 bytes long.

The path name is constructed of individual directory names and a file name separated by the forward-slash character, for example:

```
/dir1/dir2/dir3/MyFile
```

Like UNIX, z/OS UNIX is case-sensitive for file and directory names. For example, in the same directory, the file MYFILE is a different file than MyFile.

The files in a hierarchical file system are sequential files, and are accessed as byte streams. A record concept does not exist with these files other than the structure defined by an application.

The zFS data set that contains the UNIX file system is a z/OS data set type (a VSAM linear data set). zFS data sets and z/OS data sets can reside on the same DASD volume. z/OS provides commands for managing zFS space utilization.

The integration of the zFS file system with existing z/OS file system management services provides automated file system management capabilities that might not be available on other UNIX platforms. This integration allows file owners to spend less time on tasks such as backup and restore of entire file systems.

z/OS data sets versus file system files

Many elements of UNIX have analogs in the z/OS operating system. Consider, for example, that the organization of a user catalog is analogous to a user directory (/u/ibmuser) in the file system.

In z/OS, the user prefix assigned to z/OS data sets points to a user catalog. Typically, one user owns all the data sets whose names begin with his user prefix. For example, the data sets belonging to the TSO/E user ID IBMUSER all begin with the high-level qualifier (prefix) IBMUSER. There could be different data sets named IBMUSER.C, IBMUSER.C.OTHER and IBMUSER.TEST.

In the UNIX file system, ibmuser would have a user directory named /u/ibmuser. Under that directory there could be a subdirectory named /u/ibmuser/c, and /u/ibmuser/c/pgma would point to the file pgma (see Figure 16 on page 57).

Of the various types of z/OS data sets, a partitioned data set (PDS) is most like a user directory in the file system. In a partitioned data set such as IBMUSER.C, you could have members (files) PGMA, PGMB, and so on. For example, you might have IBMUSER.C(PGMA) and IBMUSER.C(PGMB). Along the same lines, a subdirectory such as /u/ibmuser/c can hold many files, such as pgma, pgmb, and so on.

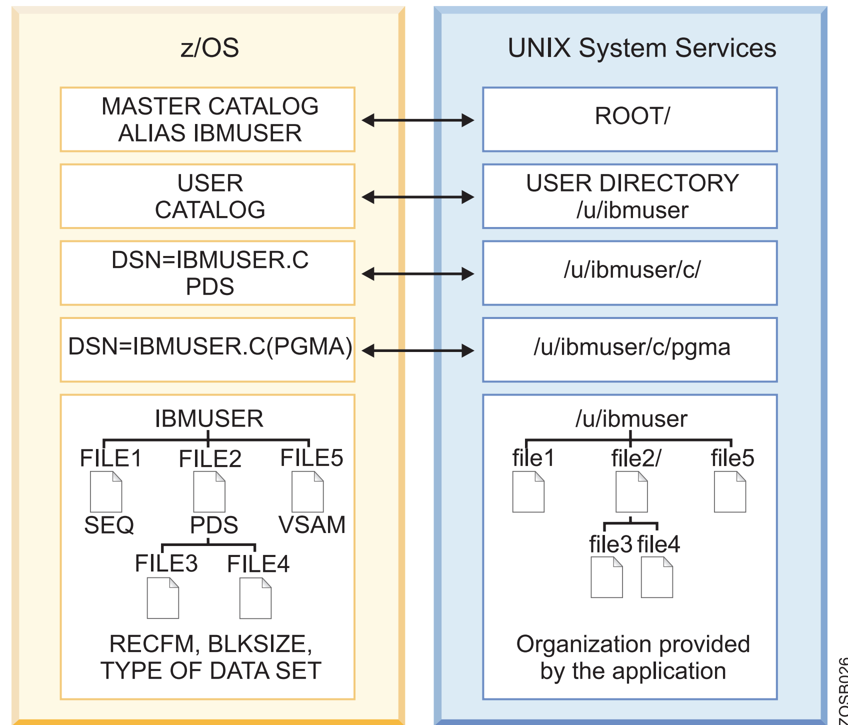


Figure 16. Comparison of z/OS data sets and file system files

All data written to a hierarchical file system can be read by all programs as soon as it is written. Data is written to a disk when a program issues an **fsync()** .

What is a zFS file system?

The z/OS Distributed File Service (DFS™) zSeries File System (zFS) is a z/OS UNIX System Services (z/OS UNIX) file system that can be used in addition to the hierarchical file system (HFS).

zFS file systems contain files and directories that can be accessed with z/OS UNIX application programming interfaces (APIs). These file systems can support access control lists (ACLs). zFS file systems can be mounted into the z/OS UNIX hierarchy along with other local (or remote) file system types (for example, HFS, TFS, AUTOMNT and NFS).

The Distributed File Service server message block (SMB) provides a server that makes z/OS UNIX files and data sets available to SMB clients. The data sets supported include sequential data sets (on DASD), PDS and PDSE, and VSAM data sets. The data set support is usually referred to as record file system (RFS) support. The SMB protocol is supported through the use of TCP/IP on z/OS. This communication protocol allows clients to access shared directory paths and shared printers. Personal computer (PC) clients on the network can use the file and print sharing functions that are included in their operating systems.

Supported SMB clients include Windows® XP Professional, Windows Terminal Server on Windows 2000 server, Windows Terminal Server on Windows 2003, and Linux®. At the same time, these files can be shared with local z/OS UNIX applications and with DCE DFS clients.

Chapter 3. Interacting with z/OS: TSO, ISPF, and z/OS UNIX interfaces

z/OS is ideal for processing batch jobs– workloads that run in the background with little or no human interaction. However, z/OS is just as much an interactive operating system as it is a batch processing system. By **interactive** we mean that end users (sometimes tens of thousands of them concurrently) can use the system through direct interaction, such as commands and menu style user interfaces.

In z/OS, the facility known as Time Sharing Option/Extensions or TSO allows multiple users to log on and interactively share the resources of the mainframe. TSO also provides users with a limited set of basic commands; using this set is sometimes called using TSO in its native mode.

ISPF is a menu-driven interface for user interaction with a z/OS system. The ISPF environment is executed from native TSO. ISPF provides utilities, an editor and ISPF applications to the user. To the extent permitted by various security controls, an ISPF user has full access to most z/OS system functions.

TSO/ISPF serves as both a system management interface and a development interface for traditional z/OS programming.

The z/OS UNIX shell and utilities provide a command interface to the z/OS UNIX environment. You can access the shell either by logging on to TSO/E or by using the remote login facilities of TCP/IP (rlogin).

If you use TSO/E, a command called OMVS creates a shell for you. You can work in the shell environment until exiting or temporarily switching back to the TSO/E environment.

What is TSO?

Time Sharing Option/Extensions (TSO/E) allows users to create an interactive session with the z/OS system. TSO provides a single-user logon capability and a basic command prompt interface to z/OS.

Most users work with TSO through its menu-driven interface, Interactive System Productivity Facility (ISPF). This collection of menus and panels offers a wide range of functions to assist users in working with data files on the system. ISPF users include system programmers, application programmers, administrators, and others who access z/OS. In general, TSO and ISPF make it easier for people with varying levels of experience to interact with the z/OS system.

In a z/OS system, each user is granted a user ID and a password authorized for TSO logon. Logging on to TSO requires a 3270 display device or, more commonly, a TN3270 emulator running on a PC.

During TSO logon, the system displays the TSO logon screen on the user’s 3270 display device or TN3270 emulator. The logon screen serves the same purpose as a Windows logon panel.

© Copyright IBM Corp. 2006, 2008

z/OS system programmers often modify the particular text layout and information of the TSO logon panel to better suit the needs of the system's users. Therefore, the screen captures used in examples will likely differ from what you might see on an actual production system.

Figure 17 shows a typical example of a TSO logon screen.

Many of the screen capture examples also show program function (PF) key settings. Because it is common practice for z/OS sites to customize the PF key assignments to suit their needs, the key assignments shown in examples might not match the PF key settings in use at your site.

```
------------------------------ TSO/E LOGON-----------------------------------


Enter LOGON parameters below:                  RACF LOGON parameters:

Userid    ===> ZUSER
Password  ===>                                 New Password  ===>

Procedure ===> IKJACCNT                        Group Ident   ===>

Acct Nmbr ===> ACCNT#

Size      ===> 860000

Perform   ===>

Command   ===>

Enter an 'S' before each option desired below:
-Nomail         -Nonotice        -Reconnect        -OIDcard

PF1/PF13 ==> Help    PF3/PF15 ==> Logoff    PA1 ==>
Attention    PA2 ==> Reshow
You may request specific help information by entering a '?' in any
entry field
```

Figure 17. Typical TSO/E logon screen

What is TSO native mode?

Most z/OS sites prefer to have the TSO user session automatically switch to the ISPF interface after TSO logon. It is possible, however, to use a limited set of basic TSO commands independent of other complementary programs, such as ISPF. Using TSO in this way is called using TSO in its native mode.

When a user logs on to TSO, the z/OS system responds by displaying the READY prompt, and waits for input, such as in Figure 18.

```
ICH70001I ZUSER   LAST ACCESS AT 17:12:12 ON THURSDAY, OCTOBER 7, 2004
ZUSER LOGON IN PROGRESS AT 17:12:45 ON OCTOBER 7, 2004
You have no messages or data sets to receive.
READY
```

Figure 18. TSO logon READY prompt

The READY prompt accepts simple line commands such as HELP, RENAME, ALLOCATE, and CALL. Figure 19 on page 61 shows an example of an ALLOCATE command that creates a data set (a file) on disk.

```
READY
 alloc dataset(zuser.test.cntl) volume(test01) unit(3390) tracks space(2,1)
recfm(f) lrecl(80) dsorg(ps)
READY
listds
 ENTER DATA SET NAME -
zuser.test.cntl
 ZUSER.TEST.CNTL
  --RECFM-LRECL-BLKSIZE-DSORG
    F     80    80      PS
 --VOLUMES--
   TEST01
READY
```

Figure 19. Allocating a data set from the TSO command line

Native TSO is similar to the interface offered by the native DOS prompt. TSO also includes a very basic line mode editor, in contrast to the full screen editor offered by ISPF.

Figure 20 is another example of the line commands a user might enter at the READY prompt. Here, the user is entering commands to sort data.

```
 READY
ALLOCATE DATASET(AREA.CODES) FILE(SORTIN)    SHR
 READY
ALLOCATE DATASET(*)          FILE(SORTOUT)   SHR
 READY
ALLOCATE DATASET(*)          FILE(SYSOUT)    SHR
 READY
ALLOCATE DATASET(*)          FILE(SYSPRINT)  SHR
 READY
ALLOCATE DATASET(SORT.CNTL)  FILE(SYSIN)     SHR
 READY
CALL `SYS1.SICELINK(SORT)'

 ICE143I 0 BLOCKSET     SORT  TECHNIQUE SELECTED
 ICE000I 1 - CONTROL STATEMENTS FOR Z/OS DFSORT V1R5
             SORT FIELDS=(1,3,CH,A)
 201  NJ
 202  DC
 203  CT
 204  Manitoba
 205  AL
 206  WA
 207  ME
 208  ID
 ***
```

Figure 20. Using native TSO commands to sort data

In this example, the user entered several TSO ALLOCATE commands to assign inputs and outputs to the workstation for the sort program. The user then entered a single CALL command to run the sort program, DFSORT™, an optional software product from IBM.

Each ALLOCATE command requires content (specified with the DATASET operand) associated with the following:

- SORTIN - in this case AREA.CODES
- SORTOUT - in this case *, which means the terminal screen
- SYSOUT
- SYSPRINT
- SYSIN

Following the input and output allocations and the user-entered CALL command, the sort program displays the results on the user's screen. As shown in Figure 20 on page 61, the SORT FIELDS control statement causes the results to be sorted by area code. For example, NJ (New Jersey) has the lowest number telephone area code, 201.

Native TSO screen control is very basic. For example, when a screen fills up with data, three asterisks (***) are displayed to indicate a full screen. Here, you must press the Enter key to clear the screen of data and allow the screen to display the remainder of the data.

How are CLISTs and REXX used?

The CLIST and REXX command languages shell script-type processing for TSO users.

With native TSO, it is possible to place a list of commands, called a **command list** or CLIST (pronounced "see list") in a file, and execute the list as if it were one command. When you invoke a CLIST, it issues the TSO/E commands in sequence. CLISTs are used for performing routine tasks; they enable users to work more efficiently with TSO.

For example, suppose that a file called AREA.COMMND contained the following sort commands:

```
ALLOCATE DATASET(AREA.CODES) FILE(SORTIN)    SHR
ALLOCATE DATASET(*)          FILE(SORTOUT)   SHR
ALLOCATE DATASET(*)          FILE(SYSOUT)    SHR
ALLOCATE DATASET(*)          FILE(SYSPRINT)  SHR
ALLOCATE DATASET(SORT.CNTL)  FILE(SYSIN)     SHR
CALL `SYS1.SICELINK(SORT)'
```

Instead of issuing each command individually, the user can achieve the same results by using just a single command to execute the CLIST, as follows:

```
EXEC 'CLIST AREA.COMMND'
```

TSO users create CLISTs with the CLIST command language. Another command language used with TSO is called Restructured Extended Executor or REXX. Both CLIST and REXX offer shell script-type processing. These command languages are **interpretive** languages, as opposed to **compiled** languages (although REXX can be compiled as well).

Some TSO users write functions directly as CLISTs or REXX programs, but these are more commonly implemented as ISPF functions, or by various software products. CLIST programming is unique to z/OS, while the REXX language is used on many platforms.

What is ISPF?

ISPF is a full panel application navigated by keyboard. ISPF includes a text editor and browser, and functions for locating and listing files and performing other utility functions.

After logging on to TSO, users typically access the ISPF menu. In fact, many use ISPF exclusively for performing work on z/OS. ISPF menus list the functions that are most frequently needed by online users.

Figure 21 shows the allocate procedure to create a data set using ISPF.

```
Menu  RefList  Utilities  Help
---------------------------------------------------------------------------
Allocate New Data Set
 Command ===>
Data Set Name  . . . : ZUSER.TEST.CNTL
Management class . . .                (Blank for default management class)
 Storage class  . . . .                (Blank for default storage class)
  Volume serial . . . . TEST01         (Blank for system default volume) **
  Device type . . . . .                (Generic unit or device address) **
 Data class . . . . . .                (Blank for default data class)
  Space units . . . . . TRACK          (BLKS, TRKS, CYLS, KB, MB, BYTES
                                        or RECORDS)
  Average record unit                  (M, K, or U)
  Primary quantity  . . 2              (In above units)
  Secondary quantity    1              (In above units)
  Directory blocks  . . 0              (Zero for sequential data set)*
  Record format . . . . F
  Record length . . . . 80
  Block size  . . . . .
  Data set name type  :                (LIBRARY, HFS, PDS, or blank)*
                                       (YY/MM/DD, YYYY/MM/DD
  Expiration date . . .                   YY.DDD, YYYY.DDD in Julian form
 Enter "/" to select option              DDDD for retention period in days
    Allocate Multiple Volumes            or blank)

( * Specifying LIBRARY may override zero directory block)

( ** Only one of these fields may be specified)
 F1=Help F2=Split F3=Exit F7=Backward F8=Forward F9=Swap F10=Actions F12=Cancel
```

Figure 21. Allocating a data set using ISPF panels

Figure 22 shows the results of allocating a data set using ISPF panels.

```
Data Set Information
Command ===>

Data Set Name  . . . : ZUSER.TEST.CNTL

General Data                          Current Allocation
 Volume serial . . . : TEST01          Allocated tracks  . : 2
 Device type . . . . : 3390            Allocated extents . : 1
 Organization  . . . : PS
 Record format . . . : F
 Record length . . . : 80
 Block size  . . . . : 80             Current Utilization
 1st extent tracks . : 2               Used tracks . . . . : 0
 Secondary tracks  . : 1               Used extents  . . . : 0

 Creation date . . . : 2005/01/31
 Referenced date . . : 2005/01/31
 Expiration date . . : ***None***

 F1=Help F2=Split F3=Exit F7=Backward  F8=Forward   F9=Swap F12=Cancel
```

Figure 22. Result of data set allocation using ISPF

Figure 23 on page 64 shows the ISPF menu structure.

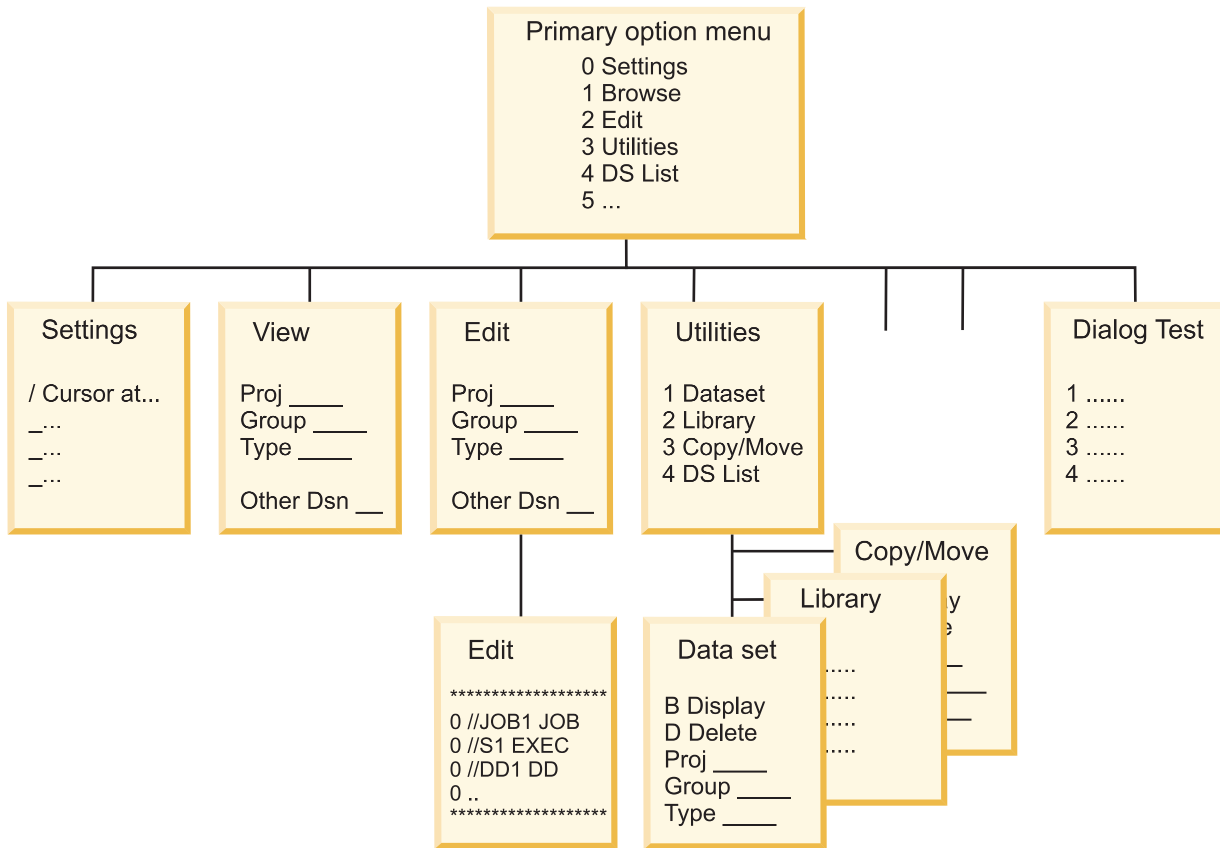


Figure 23. ISPF menu structure

To access ISPF under TSO, the user enters a command such as ISPPDF from the READY prompt to display the ISPF Primary Option Menu.

Figure 24 shows an example of the ISPF Primary Menu.

```
Menu Utilities Compilers Options  Status  Help
------------------------------------------------------------------------------
                           ISPF Primary Option Menu
Option ===>

0  Settings     Terminal and user parameters          User ID .: ZUSER
1  View         Display source data or listings       Time. . .: 17:29
2  Edit         Create or change source data          Terminal.: 3278
3  Utilities    Perform utility functions             Screen. .: 1
4  Foreground   Interactive language processing       Language.: ENGLISH
5  Batch        Submit job for language processing    Appl ID .: PDF
6  Command      Enter TSO or Workstation commands     TSO logon: IKJACCT
7  Dialog Test  Perform dialog testing                TSOprefix: ZUSER
8  LM Facility  Library administrator functions       System ID: SC04
9  IBM Products IBM program development products      MVS acct.: ACCNT#
10 SCLM         SW Configuration Library Manager      Release .: ISPF 5.2
11 Workplace    ISPF Object/Action Workplace
M  More         Additional IBM Products

Enter X to Terminate using log/list defaults

F1=Help F2=Split F3=Exit F7=Backward F8=Forward F9=Swap F10=Actions F12=Cancel
```

Figure 24. ISPF Primary Option Menu

The ISPF panel can be customized with additional options by the local system programmer. Therefore, it can vary in features and content from site to site.

To reach the ISPF menu selections shown in Figure 25, you enter `M` on the option line.

```
Menu  Help
------------------------------------------------------------------------------
                             IBM Products Panel
                                                                 More:      +
1  SMP/E         System Modification Program/Extended
2  ISMF          Integrated Storage Management Facility
3  RACF          Resource Access Control Facility
4  HCD           Hardware Configuration Dialogs
5  SDSF          Spool Search and Display Facility
6  IPCS          Interactive Problem Control System
7  DITTO         DITTO/ESA for MVS Version 1
8  RMF           Resource Measurement Facility
9  DFSORT        Data Facility Sort
10 OMVS          MVS OpenEdition
11 DB2           Data Base Products
12 RRS           Resource Recovery Services
13 DB2ADM        Data Base Admin Tool
14 QMF           Query Management Facility
15 MQ            WMQ Series Operations and Control
16 FMN           File Manager 3.1 Operations and Control
17 WLM           Workload Manager
18 PE            Performance Expert



Option ===> 9_____________________________________________________________________
 F1=Help      F2=Split      F3=Exit      F7=Backward   F8=Forward    F9=Swap
F10=Actions  F12=Cancel
```

Figure 25. More ISPF options displayed

In Figure 25, DFSORT is offered as option 9 on this panel. We will select it now as a useful example of the ISPF panel-driven applications.

Figure 26 on page 66 shows the panel that would be displayed for option 9 of ISPF.

```
DFSORT PRIMARY OPTION MENU
ENTER SELECTION OR COMMAND ===>

SELECT ONE OF THE FOLLOWING:

0  DFSORT PROFILE              - Change DFSORT user profile
1  SORT                        - Perform Sort Application
2  COPY                        - Perform Copy Application
3  MERGE                       - Perform Merge Application
X  EXIT                        - Terminate DFSORT

          \-----------------------------------------------------/
          | \-------------------------------------------------/ |
          | |       Licensed Materials - Property of IBM      | |
          | |                                                 | |
          | |  5740-SM1 (C) Copyright IBM Corp. 1988, 1992.   | |
          | |   All rights reserved.  US Government Users     | |
          | |    Restricted Rights - Use, duplication or      | |
          | |   disclosure restricted by GSA ADP Schedule     | |
          | |           Contract with IBM Corp.               | |
          | /-------------------------------------------------\ |
          /-----------------------------------------------------\

USE HELP COMMAND FOR HELP; USE END COMMAND TO EXIT.

F1=HELP     F2=SPLIT    F3=END      F4=RETURN   F5=RFIND   F6=RCHANGE
F7=UP       F8=DOWN     F9=SWAP    F10=LEFT    F11=RIGHT  F12=CURSOR
```

Figure 26. SORT panel

The TSO sort function is available through ISPF as a menu-selectable option. Through the SORT option, the user can allow ISPF to handle the TSO allocations, create the SORT control statement, and call the SORT program to produce the results of the sort.

Notice the keyboard program function key (PF key) selections at the bottom of each panel; using PF3 (END) returns the user to the previous panel.

Although ISPF is a full panel application navigated by keyboard, you may download and install a variety of ISPF graphical user interface (GUI) clients to include with a z/OS system. After installing the ISPF GUI client, it is possible to use the mouse.

ISPF keyboard keys and functions

Many screen capture examples show ISPF program function (PF) key settings at the bottom of the panel. Because it is common for z/OS users to customize the PF key assignments to suit their needs, the key assignments shown might not match the PF key settings in use on your system. Actual function key settings vary from customer to customer.

Keyboard mapping lists some of the most frequently used PF keys and other keyboard functions and their corresponding keys.

The examples in this section use these keyboard settings. For example, directions to press `Enter` mean that you should press the keyboard's control key (Ctrl) at the lower right. If the keyboard locks up, press the control key at the lower left.

Table 2. Keyboard mapping

Function	Key
Enter	Ctrl (right side)
Exit, end, or return	PF3

Table 2. Keyboard mapping (continued)

Function	Key
Help	PF1
PA1 or Attention	Alt-Ins or Esc
PA2	Alt-Home
Cursor movement	Tab or Enter
Clear	Pause
Page up	PF7
Page down	PF8
Scroll left	PF10
Scroll right	PF11
Reset locked keyboard	Ctrl (left side)

From the ISPF Primary Menu, press the PF1 HELP key to display the ISPF tutorial. New users of ISPF should acquaint themselves with the tutorial (Figure 27) and with the extensive online help facilities of ISPF.

Besides the tutorial, you can access online help from any of the ISPF panels. When you invoke help, you can scroll through information. Press the PF1-Help key for explanations of common ISPF entry mistakes, and examples of valid entries. ISPF Help also contains help for the various functions found in the primary option menu.

```
Tutorial --------------------- Table of Contents-------------------- Tutorial

                  ISPF Program Development Facility Tutorial

The following topics are presented in sequence, or may be selected by entering
a selection code in the option field:
  G  General      - General information about ISPF
  0  Settings     - Specify terminal and user parameters
  1  View         - Display source data or output listings
  2  Edit         - Create or change source data
  3  Utilities    - Perform utility functions
  4  Foreground   - Invoke language processors in foreground
  5  Batch        - Submit job for language processing
  6  Command      - Enter TSO command, CLIST, or REXX exec
  7  Dialog Test  - Perform dialog testing
  9  IBM Products - Use additional IBM program development products
  10 SCLM         - Software Configuration and Library Manager
  11 Workplace    - ISPF Object/Action Workplace
  X  Exit         - Terminate ISPF using log and list defaults
  The following topics will be presented only if selected by number:
  A  Appendices   - Dynamic allocation errors and ISPF listing formats
  I  Index        - Alphabetical index of tutorial topics

F1=Help      F2=Split     F3=Exit      F4=Resize    F5=Exhelp   F6=Keyshelp
F7=PrvTopic  F8=NxtTopic  F9=Swap     F10=PrvPage  F11=NxtPage F12=Cancel
```

Figure 27. ISPF Tutorial main menu

PA1 is a very important key for TSO users and every user should know how to find it on the keyboard. Back in the early days, the "real" 3270 terminals had keys labeled PA1, PA2, and PA3. These were called Program Action keys or PA keys. In practice, only PA1 is still widely used and it functions as a break key for TSO. In TSO terminology, this is an attention interrupt. That is, pressing the PA1 key will end the current task.

Finding the PA1 key on the keyboard of a 3270 terminal emulator such as TN3270 emulator can be a challenge. A 3270 emulator can be customized to many different key combinations. On an unmodified x3270 session, the PA1 key is Left Alt-1.

The ISPF Data Set List utility

z/OS users typically use the ISPF Data Set List utility to work with data sets.

To access this utility from the ISPF Primary Option Menu, select Utilities, then select Dslist to display the Utility Selection Panel, which is shown in Figure 28.

```
Menu  RefList  RefMode  Utilities  Help
------------------------------------------------------------------------------
                           Data Set List Utility
Option ===> ____________________________________________________________

  blank Display data set list              P Print data set list
      V Display VTOC information          PV Print VTOC information

Enter one or both of the parameters below:
  Dsname Level . . .  ZUSER_______________________________
   Volume serial  . .  ______
Data set list options
   Initial View . . . 1  1. Volume       Enter "/" to select option
                         2. Space        /  Confirm Data Set Delete
                         3. Attrib       /  Confirm Member Delete
                         4. Total        /  Include Additional Qualifiers

When the data set list is displayed, enter either:
  "/" on the data set list command field for the command promptpop-up,
  an ISPF line command, the name of a TSO command, CLIST, or REXX exec, or
  "=" to execute the previous command.

F1=Help F2=Split F3=Exit F7=Backward  F8=Forward F9=Swap F10=Actions  F12=Cancel
```

Figure 28. Using the Data Set List utility

In the panel, you can use the Dsname Level data entry field to locate and list data sets. To search for one data set in particular, enter the complete (or **fully qualified**) data set name. To search for a range of data sets, such as all data sets sharing a common HLQ, enter only the HLQ in the Dsname Level field.

Qualifiers can be specified fully, partially, or defaulted. At least one qualifier must be partially specified. To search for a portion of a name, specify an asterisk (*) before or after part of a data set name. Doing so will cause the utility to return all data sets that match the search criteria. Avoid searching on * alone, because TSO has many places to search in z/OS so this could take quite awhile.

In the majority of ISPF panels, a fully qualified data set name needs to be enclosed in single quotes. Data set names not enclosed in single quotes will, by default, be prefixed with a high level qualifier specified in the TSO PROFILE. This default can be changed using the PROFILE PREFIX command. In addition, an exception is ISPF option 3.4 DSLIST; do not enclose Dsname Level in quotes on this panel.

For example, if you enter `ZUSER` in the Dsname field, the utility lists all data sets with **ZUSER** as a high-level qualifier. The resulting list of data set names (see Figure 29 on page 69) allows the user to edit or browse the contents of any data set in the list.

```
Menu  Options  View  Utilities  Compilers  Help
------------------------------------------------------------------------------
DSLIST - Data Sets Matching ZUSER                          Row 1 of 4
Command ===>                                                  Scroll ===> PAGE

Command - Enter "/" to select action                  Message         Volume
-------------------------------------------------------------------------------
       ZUSER                                                    *ALIAS
       ZUSER.JCL.CNTL                                           SMITH1
       ZUSER.LIB.SOURCE                                         SMITH1
       ZUSER.PROGRAM.CNTL                                       SMITH1
       ZUSER.PROGRAM.LOAD                                       SMITH1
       ZUSER.PROGRAM.SRC                                        SMITH1
***************************** End of Data Set list****************************


F1=Help F2=Split F3=Exit F5=Rfind F7=Up F8=Down F9=Swap F10=Left F11=Right F12=Cancel
```

Figure 29. Data Set List results for dsname ZUSER

To see all of the possible actions you might take for a given data set, specify a forward slash (/) in the command column to the left of the data set name. ISPF will display a list of possible actions, as shown in Figure 30.

```
Menu  Options  View  Utilities  Compilers  Help
- +-------------------------------------------------------------+----------
D !                  Data Set List Actions                      ! Row 1 of 4
C !                                                             !  ===> PAGE
  ! Data Set: ZUSER.PROGRAM.CNTL                                !
C !                                                             !    Volume
- ! DSLIST Action                                               !-----------
  ! __  1.  Edit                    12. Compress                !    *ALIAS
/ !     2.  View                    13. Free                    !    SMITH1
  !     3.  Browse                  14. Print Index             !    SMITH1
  !     4.  Member List             15. Reset                   !    SMITH1
* !     5.  Delete                  16. Move                    !***********
  !     6.  Rename                  17. Copy                    !
  !     7.  Info                    18. Refadd                  !
  !     8.  Short Info              19. Exclude                 !
  !     9.  Print                   20. Unexclude 'NX'          !
  !     10. Catalog                 21. Unexclude first 'NXF'   !
  !     11. Uncatalog               22. Unexclude last 'NXL'    !
  !                                                             !
  ! Select a choice and press ENTER to process data set action. !
  !  F1=Help        F2=Split       F3=Exit        F7=Backward   !
  !  F8=Forward     F9=Swap       F12=Cancel                    !
  +-------------------------------------------------------------+

F1=Help F2=Split F3=Exit F5=Rfind F7=Up F8=Down F9=Swap F10=Left F11=Right F12=Cancel
```

Figure 30. Displaying the Data Set List actions

The ISPF editor

z/OS users typically use the ISPF editor to create or modify data set members.

To access the editor, select 2 from the ISPF Primary Option Menu and, on the Edit Entry panel, enter the name of the data set that you want to create or modify.

In edit mode, each line of text in the data set is known as a **record**. You can perform the following tasks:

- To edit the contents of a data set, move the cursor to the area of the record to be changed and type over the existing text.
- To find and change text, you can enter commands on the editor command line.

- To insert, copy, delete, or move text, place these commands directly on the line numbers where the action should occur.

To commit your changes, use PF3 or save. To exit the data set without saving your changes, enter `Cancel` on the edit command line.

Figure 31 shows the contents of data set **ZUSER**.PROGRAM.CNTL(SORTCNTL) opened in edit mode.

```
File   Edit   Edit_Settings   Menu   Utilities   Compilers   Test   Help
----------------------------------------------------------------------------

EDIT  ZUSER.PROGRAM.CNTL(SORTCNTL) - 01.00  Columns 00001 00072

Command ===>                      Scroll ===> CSR

**************************** Top of Data *****************************

000010 SORT FIELDS=(1,3,CH,A)

*************************** Bottom of Data ***************************
```

Figure 31. Edit a data set

Take a look at the line numbers, the text area, and the editor command line. Primary command line, line commands placed on the line numbers, and text overtype are three different ways in which you can modify the contents of the data set. Line numbers increment by 10 with the TSO editor so that the programmer can insert nine additional lines between each current line without having to renumber the program.

PF1 in edit mode displays the entire editor tutorial (Figure 32).

```
TUTORIAL -------------------------- EDIT ----------------------------- TUTORIAL
OPTION  ===>

                     -----------------------------------
                     |             EDIT                |
                     -----------------------------------

  Edit allows you to create or change source data.

The following topics are presented in sequence, or may be selected by number:
  0 - General introduction             8 - Display modes(CAPS/HEX/NULLS)
  1 - Types of data sets               9 - Tabbing (hardware/software/logical)
  2 - Edit entry panel                10 - Automatic recovery
  3 - SCLM edit entry panel           11 - Edit profiles
  4 - Member selection list           12 - Edit line commands
  5 - Display screen format           13 - Edit primary commands
  6 - Scrolling data                  14 - Labels and line ranges
  7 - Sequence numbering              15 - Ending an edit session

The following topics will be presented only if selected by number:
 16 - Edit models
 17 - Miscellaneous notes about edit


F1=Help      F2=Split     F3=Exit      F4=Resize    F5=Exhelp   F6=Keyshelp
F7=PrvTopic  F8=NxtTopic  F9=Swap     F10=PrvPage  F11=NxtPage F12=Cancel
```

Figure 32. Edit Help panel and tutorial

A subset of the line commands includes:

i Insert a line

Enter Press Enter without entering anything to escape insert mode

i5 Obtain five input lines
d Delete a line
d5 Delete five lines
dd/dd Delete a block of lines
r Repeat a line
rr/rr Repeat a block of lines
c With **a** or **b**: Copy a line after or before
c5 With **a** or **b**: Copy five lines after or before
cc/cc With **a** or **b**: Copy a block of lines after or before
m Move lines. As with copy commands, **m5**, **mm/mm** and use with **a** or **b** are also valid commands.
x Exclude a line

The ISPF Settings menu

The ISPF Settings menu and HILITE command allow you to customize the appearance of your ISPF session.

To access and change ISPF settings, do the following:

1. From the ISPF Primary Option Menu, select option 0 to display the Settings menu, as shown in Figure 33.
2. In the list of options, type or remove the "/" on the line corresponding to the setting that you want to change. Use the Tab or New line key to move the cursor.

```
Log/List  Function keys  Colors  Environ  Workstation  Identifier  Help
---------------------------------------------------------------------------
                                ISPF Settings
Command ===>

Options                                   Print Graphics
Enter "/" to select option                  Family printer type 2
_  Command line at bottom                   Device name . . . .
/  Panel display CUA mode                   Aspect ratio  . . . 0
/  Long message in pop-up
_  Tab to action bar choices
_  Tab to point-and-shoot fields         General
/  Restore TEST/TRACE options               Input field pad . . B
_  Session Manager mode                     Command delimiter . ;
/  Jump from leader dots
_  Edit PRINTDS Command
/  Always show split line
_  Enable EURO sign

Terminal Characteristics
  Screen format   2  1. Data    2. Std     3. Max     4. Part

  Terminal Type   3    1. 3277       2. 3277A      3. 3278       4. 3278A
                       5. 3290A      6. 3278T      7. 3278CF     8. 3277KN
                       9. 3278KN    10. 3278AR    11. 3278CY    12. 3278HN
                      13. 3278HO    14. 3278IS    15. 3278L2    16. BE163
                      17. BE190     18. 3278TH    19. 3278CU    20. DEU78
                      21. DEU78A    22. DEU90A    23. SW116     24. SW131
                      25. SW500
```

Figure 33. ISPF settings

The actions in the bar across the top usually vary from site to site.

Another way to customize ISPF panels is with the **hilite** command, as shown in Figure 34 on page 72. This command allows you to tailor various ISPF options to

suit the needs of your environment.

```
File  Languages  Colors  Help
------------------------------------------------------------------------------
                             Edit Color Settings
Command ===> (this menu shows up when you type "hilite")

Language: __ 1. Automatic     Coloring:  __ 1. Do not color program
             2. Assembler                   2. Color program
             3. BookMaster                  3. Both IF and DO logic
             4. C                           4. DO logic only
             5. COBOL                       5. IF logic only
             6. IDL
             7. ISPF DTL         Enter "/" to select option
             8. ISPF Panel       ___ Parentheses matching
             9. ISPF Skeleton    _/_ Highlight FIND strings
            10. JCL              _/_ Highlight cursor phrase
            11. Pascal
            12. PL/I             Note: Information from this...
            13. REXX             saved in the edit profile.

F1=Help    F2=Split      F3=Exit    F7=Backward F8=Forward
F9=Swap    F10=Actions  F12=Cancel
```

Figure 34. Using the HILITE command

What is z/OS UNIX?

The z/OS UNIX shell and utilities provide an interactive interface to z/OS.

The shell and utilities can be compared to the TSO function in z/OS.

To perform some command requests, the shell calls other programs, known as **utilities**. The shell can be used to:

- Invoke shell scripts and utilities.
- Write shell scripts (a named list of shell commands, using the shell programming language).
- Run shell scripts and C language programs interactively, in the TSO background or in batch.

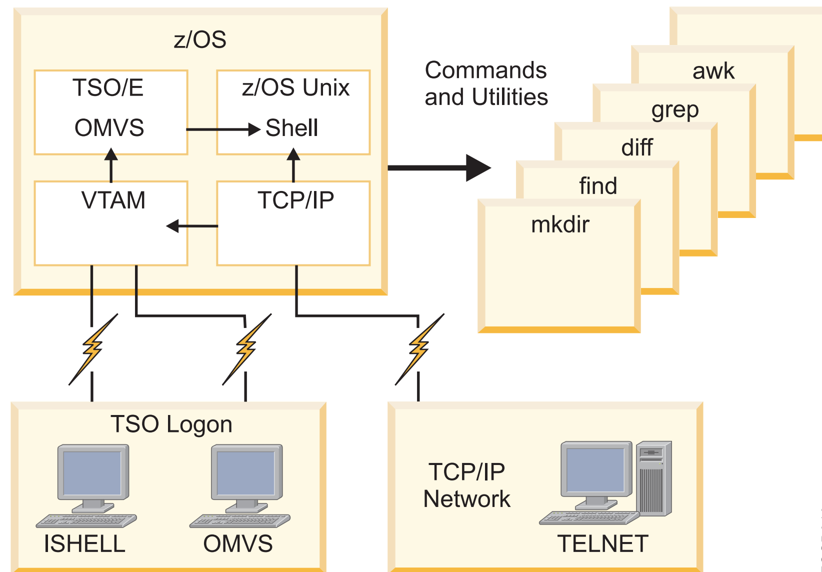


Figure 35. Shell and utilities

A user can invoke the z/OS UNIX shell in the following ways:

- From a 3270 display or a workstation running a 3270 emulator
- From a TCP/IP-attached terminal, using the **rlogin** and **telnet** commands
- From a TSO session, using the OMVS command.

As an alternative to invoking the shell directly, a user can use ISHELL by entering the command ISHELL from TSO. ISHELL provides an ISPF panel interface to perform many actions for z/OS UNIX operations.

Figure 36 shows an overview of these interactive interfaces, the z/OS UNIX shell and ISHELL. Also, there are some TSO/E commands that support z/OS UNIX, but they are limited to functions such as copying files and creating directories.

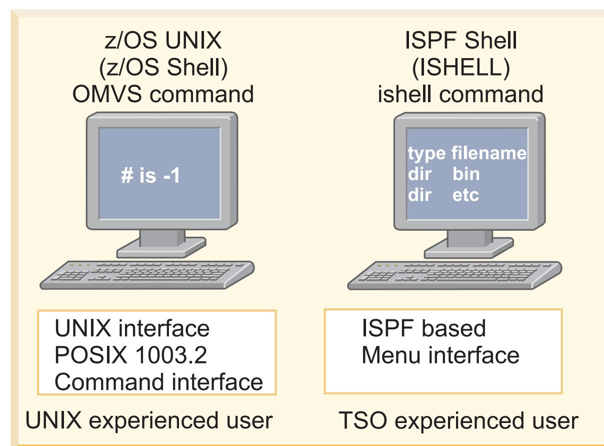


Figure 36. z/OS UNIX interactive interfaces

The z/OS UNIX shell is based on the UNIX System V shell and has some of the features from the UNIX Korn shell. The POSIX standard distinguishes between a **command**, which is a directive to the shell to perform a specific task, and a **utility**,

which is the name of a program callable by name from the shell. To the user, there is no difference between a command and a utility.

The z/OS UNIX shell provides the environment that has the most functions and capabilities. It supports many of the features of a regular programming language.

You can store a sequence of shell commands in a text file that can be executed. This is called a **shell script**.

The TSO commands used with z/OS UNIX are:

ISHELL
: The ISHELL command invokes the ISPF panel interface to z/OS UNIX System Services. ISHELL is a good starting point for users familiar with TSO and ISPF who want to use z/OS UNIX. These users can do much of their work with ISHELL, which provides panels for working with the z/OS UNIX file system, including panels for mounting and unmounting file systems and for doing some z/OS UNIX administration. ISHELL is often good for system programmers, familiar with z/OS, who need to set up UNIX resources for the users.

OMVS
: The OMVS command is used to invoke the z/OS UNIX shell. Users whose primary interactive computing environment is a UNIX system should find the z/OS UNIX shell environment familiar.

ISHELL command (ish)

The ISHELL command invokes the ISPF panel interface to z/OS UNIX System Services. ISHELL is a good starting point for users familiar with TSO and ISPF who want to use z/OS UNIX.

Figure 37 shows the ISHELL or ISPF Shell panel displayed as a result of the ISHELL or ISH command being entered from ISPF Option 6.

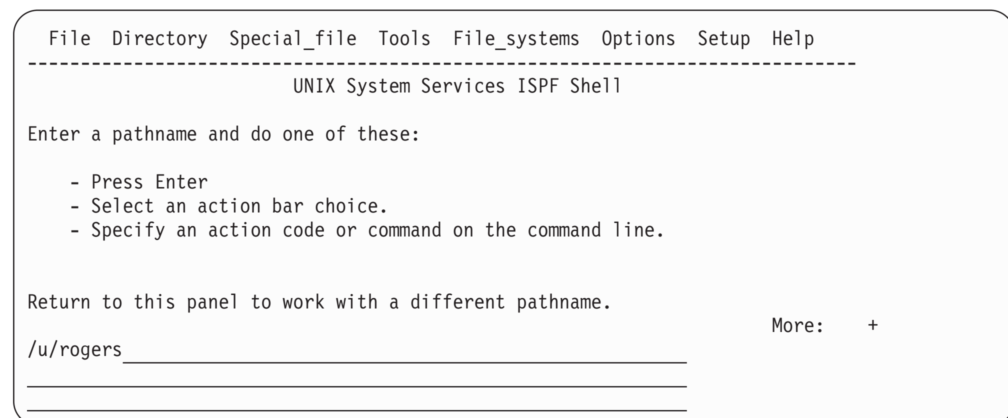


Figure 37. Panel displayed after issuing the ISH command

To search a user's files and directories, type `/u/userid` and then press `Enter`. or example, Figure 38 on page 75 shows the files and directories of user **rogers**.

```
----------------------------------------------------------------------------
                         Directory List

Select one or more files with / or action codes.  If / is used also select an
action from the action bar otherwise your default action will be used.  Select
with S to use your default action.  Cursor select can also be used for quick
navigation.  See help for details.
EUID=0   /u/rogers
  Type  Perm  Changed-EST5EDT   Owner        -Size  Filename   Row 1 of 9
 _Dir    700  2002-08-01 10:51  ADMIN         8129  .
 _Dir    555  2005-02-13 11:14  AAAAAAA          0  ..
 _File   755  1996-02-29 18:02  ADMIN          979  .profile
 _File   600  1996-03-01 10:29  ADMIN           29  .sh_history
 _Dir    755  2001-06-25 17:43  AAAAAAA       8129  data
 _File   644  2004-06-26 11:27  AAAAAAA      47848  inventory.export
 _File   700  2002-08-01 10:51  AAAAAAA         16  myfile
 _File   644  2007-06-22 17:53  AAAAAAA      43387  print.export
 _File   644  2007-04-28 18:03  AAAAAAA      84543  Sc.pdf
```

Figure 38. Display of a user's files and directories

From here, you use action codes to do any of the following:

b Browse a file or directory
e Edit a file or directory
d Delete a file or directory
r Rename a file or directory
a Show the attributes of a file or directory
c Copy a file or directory

OMVS command shell session

The OMVS command is used to invoke the z/OS UNIX shell. Users whose primary interactive computing environment is a UNIX system should find the z/OS UNIX shell environment familiar.

You use the OMVS command to invoke the z/OS UNIX shell.

The shell is a command processor that you use to:

- Invoke shell commands or utilities that request services from the system.
- Write shell scripts using the shell programming language.
- Run shell scripts and C-language programs interactively (in the foreground), in the background, or in batch.

Shell commands often have options (also known as **flags**) that you can specify, and they usually take an argument, such as the name of a file or directory. The format for specifying the command begins with the command name, then the option or options, and finally the argument, if any.

For example, in Figure 39 on page 76 the following command is shown:

```
ls -al /u/rogers
```

where **ls** is the command name, and **-al** are the options.

```
ROGERS @ SC43:/>ls -al /u/rogers
total 408
drwx------  3 ADMIN   SYS1          8192 Aug  1 2005  .
dr-xr-xr-x 93 AAAAAAA TTY              0 Feb 13 11:14 ..
-rwxr-xr-x  1 ADMIN   SYS1           979 Feb 29 1996  .profile
-rw-------  1 ADMIN   SYS1            29 Mar  1 1996  .sh_history
-rw-r--r--  1 AAAAAAA SYS1         84543 Apr 28 2007  Sc.pdf
drwxr-xr-x  2 AAAAAAA SYS1          8192 Jun 25 2001  data
-rw-r--r--  1 AAAAAAA SYS1         47848 Jun 26 2004  inventory.export
-rwx------  1 AAAAAAA SYS1            16 Aug  1 2005  myfile
-rw-r--r--  1 AAAAAAA SYS1         43387 Jun 22 2007  print.export
```

Figure 39. OMVS shell session display after issuing the OMVS command

This command lists the files and directories of the user. If the pathname is a file, **ls** displays information on the file according to the requested options. If it is a directory, **ls** displays information on the files and subdirectories therein. You can get information on a directory itself by using the **-d** option.

If you do not specify any options, **ls** displays only the file names. When **ls** sends output to a pipe or file, it writes one name per line; when it sends output to the terminal, it uses the **-C** (multi-column) format.

Terminology note: z/OS users tend to use the terms **data set** and **file** synonymously, but not when it comes to z/OS UNIX System Services. With the UNIX support in z/OS, the file system is a data set that contains directories and files. So file has a very specific definition. z/OS UNIX files are different from other z/OS data sets because they are byte-oriented rather than record-oriented.

Direct login to the z/OS UNIX shell

You can log in directly to the z/OS UNIX shell from a system that is connected to z/OS through TCP/IP.

Use one of the following methods:

rlogin You can rlogin (remote log in) to the shell from a system that has rlogin client support. To log in, use the rlogin command syntax supported at your site.

telnet You can telnet into the shell. To log in, use the telnet command from your workstation or from another system with telnet client support.

As shown in Figure 40 on page 77, each of these methods requires the **inetd daemon** to be set up and active on the z/OS system.

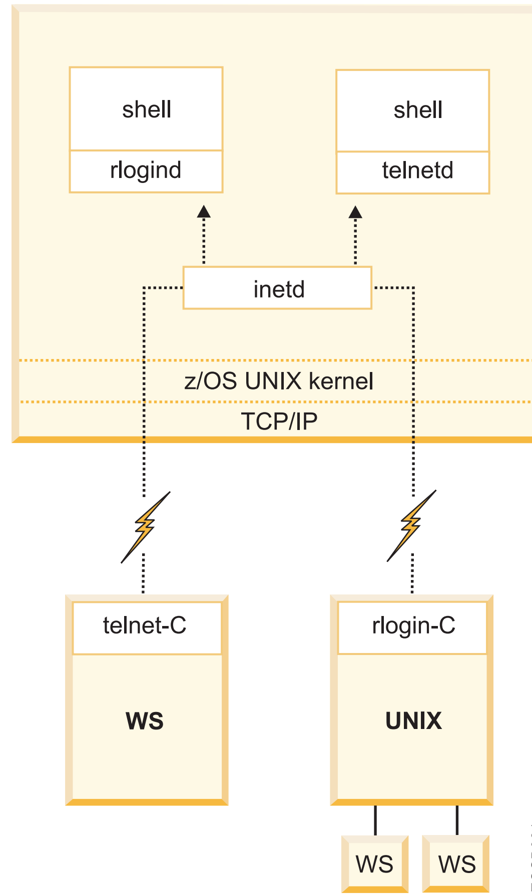


Figure 40. Diagram of a login to the shell from a terminal

There are some differences between the asynchronous terminal support (direct shell login) and the 3270 terminal support (OMVS command):

- You cannot switch to TSO/E. However, you can use the TSO SHELL command to run a TSO/E command from your shell session.
- You cannot use the ISPF editor (this includes the **oedit** command, which invokes ISPF edit).
- You can use the UNIX vi editor, and other interactive utilities that depend on receiving each keystroke, without hitting the Enter key.
- You can use UNIX-style command-line editing.

Chapter 4. Processing work on z/OS: How the system starts and manages batch jobs

Your company's core applications, such as payroll, are usually performed through batch processing, which involves executing one or more batch jobs in a sequential flow. The job entry subsystem (JES) helps z/OS receive jobs, schedule them for processing, and determine how job output is processed.

Batch processing is the most fundamental function of z/OS. Many batch jobs are run in parallel and JCL is used to control the operation of each job. Correct use of JCL parameters (especially the DISP parameter in DD statements) allows parallel, asynchronous execution of jobs that may need access to the same data sets.

An **initiator** is a system program that processes JCL, sets up the necessary environment in an address space, and runs a batch job in the same address space. Multiple initiators (each in an address space) permit the parallel execution of batch jobs.

One goal of an operating system is to process work while making the best use of system resources. To achieve this goal, resource management is needed during key phases to do the following:

- Before job processing, reserve input and output resources for jobs.
- During job processing, manage spooled SYSIN and SYSOUT data.
- After job processing, free all resources used by the completed jobs, making the resources available to other jobs.

z/OS shares with the **job entry subsystem** (JES) the management of jobs and resources. JES receives jobs into the system, schedules them for processing by z/OS, and controls their output processing. JES is the manager of the jobs waiting in a queue. It manages the priority of the jobs and their associated input data and output results. The initiator uses the statements in the JCL records to specify the resources required of each individual job after it is released (dispatched) by JES.

IBM provides two kinds of job entry subsystems: JES2 and JES3. In many cases, JES2 and JES3 perform similar functions.

During the life of a job, both JES and the z/OS base control program control different phases of the overall processing. Jobs are managed in queues: Jobs that are waiting to run (conversion queue), currently running (execution queue), waiting for their output to be produced (output queue), having their output produced (hard-copy queue), and waiting to be purged from the system (purge queue).

What is batch processing?

Jobs that can run without end user interaction, or can be scheduled to run as resources permit, are called batch jobs. Batch processing is for those frequently used programs that can be executed with minimal human interaction.

A program that reads a large file and generates a report, for example, is considered to be a batch job.

© Copyright IBM Corp. 2006, 2008

The term **batch job** originated in the days when punched cards contained the directions for a computer to follow when running one or more programs. Multiple card decks representing multiple jobs would often be stacked on top of one another in the hopper of a card reader, and be run in batches.

As a historical note, Herman Hollerith (1860-1929) created the punched card in 1890 while he worked as a statistician for the United States Census Bureau. To help tabulate results for the 1890 U.S. census, Hollerith designed a paper card with 80 columns and 12 rows; he made it equal to the size of a U.S. dollar bill of that time. To represent a series of data values, he punched holes into the card at the appropriate row/column intersections. Hollerith also designed an electromechanical device to "read" the holes in the card, and the resulting electrical signal was sorted and tabulated by a computing device. (Mr. Hollerith later founded the Computing Tabulating Recording Company, which eventually became IBM.)

There is no direct counterpart to z/OS batch processing in PC or UNIX systems. Batch jobs are typically executed at a scheduled time or on an as-needed basis. Perhaps the closest comparison is with processes run by an AT® or CRON command in UNIX, although the differences are significant. You might also consider batch processing as being somewhat analogous to the printer queue as it is typically managed on an Intel-based operating system. Users submit jobs to be printed, and the print jobs wait to be processed until each is selected by priority from a queue of work called a print spool.

To enable the processing of a batch job, z/OS professionals use job control language (JCL) to tell z/OS which programs are to be executed and which files will be needed by the executing programs. JCL allows the user to describe certain attributes of a batch job to z/OS, such as:

- Who you are (the submitter of the batch job)
- What program to run
- Where input and output are located
- When a job is to run

After the user submits the job to the system, there is normally no further human interaction with the job until it is complete.

What is JES?

z/OS uses a **job entry subsystem** or JES to receive jobs into the operating system, to schedule them for processing by z/OS, and to control their output processing.

JES is the component of the operating system that provides supplementary job management, data management, and task management functions such as scheduling, control of job flow, and the reading and writing of input and output streams on auxiliary storage devices, concurrently with job execution.

z/OS manages work as tasks and subtasks. Both transactions and batch jobs are associated with an internal task queue that is managed on a priority basis. JES is a component of z/OS that works on the front end of program execution to prepare work to be executed. JES is also active on the back end of program execution to help clean up after work is performed. This activity includes managing the printing of output generated by active programs.

More specifically, JES manages the input and output job queues and data.

For example, JES handles the following aspects of batch processing for z/OS:

- Receiving jobs into the operating system
- Scheduling them for processing by z/OS
- Controlling their output processing

z/OS has two versions of job entry systems: JES2 and JES3. Of these, JES2 is the most common by far. JES2 and JES3 have many functions and features, but their most basic functions are as follows:

- Accept jobs submitted in various ways:
  - From ISPF through the SUBMIT command
  - Over a network
  - From a running program, which can submit other jobs through the JES internal reader
  - From a card reader (very rare!)
- Queue jobs waiting to be executed. Multiple queues can be defined for various purposes.
- Queue jobs for an **initiator**, which is a system program that requests the next job in the appropriate queue.
- Accept printed output from a job while it is running and queue the output.
- Optionally, send output to a printer, or save it on **spool** for PSF, InfoPrint, or another output manager to retrieve.

JES uses one or more disk data sets for **spooling**, which is the process of reading and writing input and output streams on auxiliary storage devices, concurrently with job execution, in a format convenient for later processing or output operations. **Spool** is an acronym that stands for **simultaneous peripheral operations online**.

JES combines multiple spool data sets (if present) into a single conceptual data set. The internal format is not in a standard access-method format and is not written or read directly by applications. Input jobs and printed output from many jobs are stored in the single (conceptual) spool data set. In a small z/OS system, the spool data sets might be a few hundred cylinders of disk space; in a large installation, they might be many complete volumes of disk space.

The basic elements of batch processing are shown in Figure 41 on page 82.

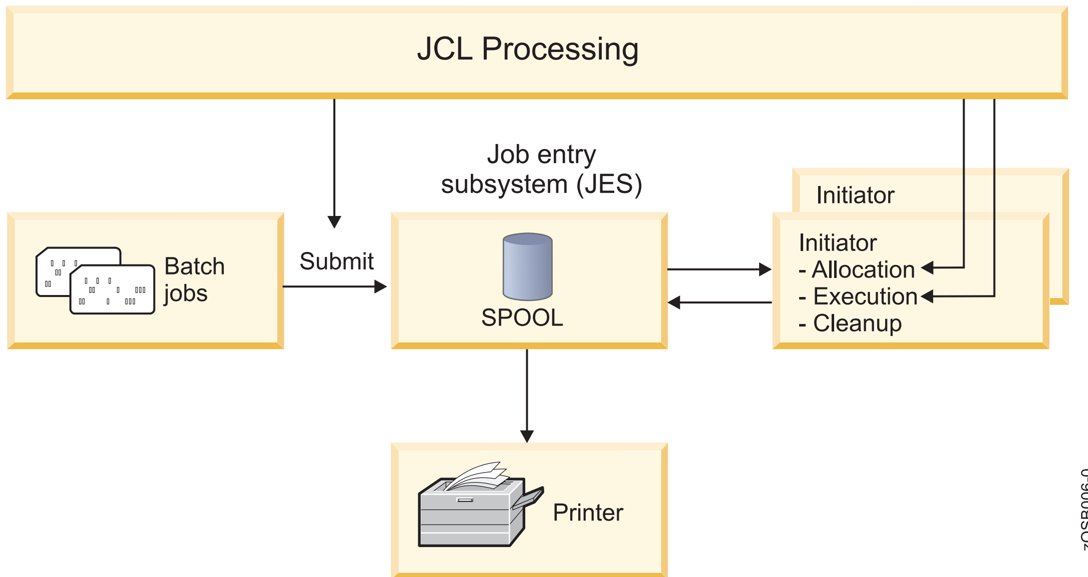


Figure 41. Basic batch flow

The initiator is an integral part of z/OS that reads, interprets, and executes the JCL. It is normally running in several address spaces (as multiple initiators). An initiator manages the running of batch jobs, one at a time, in the same address space. If ten initiators are active (in ten address spaces), then ten batch jobs can run at the same time. JES does some JCL processing, but the initiator does the key JCL work

The jobs in Basic batch flow represent JCL and perhaps data intermixed with the JCL. Source code input for a compiler is an example of data (the source statements) that might be intermixed with JCL. Another example is an accounting job that prepares the weekly payroll for different divisions of a firm (presumably, the payroll application program is the same for all divisions, but the input and master summary files may differ).

The diagram represents the jobs as punched cards (using the conventional symbol for punched cards) although real punched card input is very rare now. Typically, a job consists of card images (80-byte fixed-length records) in a member of a partitioned data set.

What does an initiator do?

An initiator performs several functions to ensure that multiple jobs run at the same time, without conflicts.

To run multiple jobs asynchronously, the system must perform a number of functions:

- Select jobs from the input queues (JES does this).
- Ensure that multiple jobs (including TSO users and other interactive applications) do not conflict in data set usage.
- Ensure that single-user devices, such as tape drives, are allocated correctly.
- Find the executable programs requested for the job.
- Clean up after the job ends and then request the next job.

Most of this work is done by the initiator, based on JCL information for each job. The most complex function is to ensure there are no conflicts due to data set utilization. For example, if two jobs try to write in the same data set at the same time (or one reads while the other writes), there is a conflict. This event would normally result in corrupted data. The primary purpose of JCL is to tell an initiator what is needed for the job.

The prevention of conflicting data set usage is critical to z/OS and is one of the defining characteristics of the operating system. When the JCL is properly constructed, the prevention of conflicts is automatic. For example, if job A and job B must both write to a particular data set, the system (through the initiator) does not permit both jobs to run at the same time. Instead, whichever job starts first causes an initiator attempting to run the other job to wait until the first job completes.

Batch processing and JES: Scenario 1

Imagine that you are a z/OS application programmer developing a program for non-skilled users. Your program is supposed to read a couple of files, write to another couple of files, and produce a printed report. This program will run as a batch job on z/OS.

What sorts of functions are needed in the operating system to fulfill the requirements of your program? And, how will your program access those functions?

First, you need a sort of special language to inform the operating system about your needs. On z/OS, this language is job control language (JCL). JCL provides the means for you to request resources and services from the operating system for a batch job.

Specifications and requests you might make for a batch job include the functions you need to compile and execute the program, and allocate storage for the program to use as it runs.

With JCL, you can specify the following:

- Who you are (important for security reasons).
- Which resources (programs, files, memory) and services are needed from the system to process your program. You might, for example, need to do the following:
  - Load the compiler code in memory.
  - Make accessible to the compiler your source code; that is, when the compiler asks for a read, your source statements are brought to the compiler memory.
  - Allocate some amount of memory to accommodate the compiler code, I/O buffers, and working areas.
  - Make accessible to the compiler an output disk data set to receive the object code, which is usually referred to as the object deck or simply OBJ.
  - Make accessible to the compiler a print file where it will tell you your eventual mistakes.
  - Conditionally, have z/OS load the newly created object deck into memory (but skip this step if the compilation failed).
  - Allocate some amount of memory for your program to use.
  - Make accessible to your program all the input and output files.

– Make accessible to your program a printer for eventual messages.

In turn, you require the operating system to:

- Convert JCL to control blocks that describe the required resources.
- Allocate the required resources (programs, memory, files).
- Schedule the execution on a timely basis; for example, your program only runs if the compilation succeeds.
- Free the resources when the program is done.

The parts of z/OS that perform these tasks are the job entry subsystem (JES) and a batch initiator program.

Think of JES as the manager of the jobs waiting in a queue. It manages the priority of the set of jobs and their associated input data and output results. The initiator uses the statements on the JCL cards to specify the resources required of each individual job once it has been released (dispatched) by JES.

Your JCL as described is called a **job**, in this case formed by two sequential steps, the compilation and execution. The steps in a job are always executed sequentially. The job must be submitted to JES to be executed. To make your task easier, z/OS provides a set of procedures in a data set called SYS1.PROCLIB. A procedure is a set of JCL statements that are ready to be executed.

Figure 42 on page 85 shows a JCL procedure that can compile, link-edit and execute a program. The first step identifies the COBOL compiler, as declared in **//COBOL EXEC PGM=IGYCRCTL** . The statement **//SYSLIN DD** describes the output of the compiler (the object deck).

The object deck is the input for the second step, which performs link-editing (through program IEWL). Link-editing is needed to resolve external references and **bring in** or **link** the previously developed common routines (a type of code re-use).

In the third step, the program is executed.

```
000010 //IGYWCLG PROC LNGPRFX='IGY.V3R2M0',SYSLBLK=3200,
000020 //             LIBPRFX='CEE',GOPGM=GO
000030 //*
000040 //********************************************************************
000050 //*                                                                  *
000060 //*  Enterprise COBOL for z/OS and OS/390                            *
000070 //*               Version 3 Release 2 Modification 0                 *
000080 //*                                                                  *
000090 //*  LICENSED MATERIALS - PROPERTY OF IBM.                           *
000100 //*                                                                  *
000110 //*  5655-G53 5648-A25 (C) COPYRIGHT IBM CORP. 1991, 2002            *
000120 //*  ALL RIGHTS RESERVED                                             *
000130 //*                                                                  *
000140 //*  US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                    *
000150 //*  DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                     *
000160 //*  ADP SCHEDULE CONTRACT WITH IBM CORP.                            *
000170 //*                                                                  *
000180 //********************************************************************
000190 //*
000300 //COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
000310 //STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
000320 //             DISP=SHR
000330 //SYSPRINT DD  SYSOUT=*
000340 //SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
000350 //             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
000360 //             DCB=(BLKSIZE=&SYSLBLK)
000370 //SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
000440 //LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
000450 //SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,
000460 //             DISP=SHR
000470 //SYSPRINT DD  SYSOUT=*
000480 //SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
000490 //         DD  DDNAME=SYSIN
000500 //SYSLMOD  DD  DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
000510 //             UNIT=SYSDA,DISP=(MOD,PASS)
000520 //SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
000530 //GO     EXEC  PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
000540 //             REGION=2048K
000550 //STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
000560 //             DISP=SHR
000570 //SYSPRINT DD  SYSOUT=*
000580 //CEEDUMP  DD  SYSOUT=*
000590 //SYSUDUMP DD  SYSOUT=*
```

Figure 42. Procedure to compile, link-edit, and execute programs

To invoke a procedure, you can write some simple JCL, as shown in Figure 43 on page 86. In this example, we added other DD statements, such as//`COBOL.SYSIN DD *`, which identifies the data set that contains the COBOL source code.

```
000001 //COBOL1 JOB (POK,999),MGELINSKI,MSGLEVEL=(1,1),MSGCLASS=X,
000002 //  CLASS=A,NOTIFY=&SYSUID
000003 /*JOBPARM SYSAFF=*
000004 // JCLLIB   ORDER=(IGY.SIGYPROC)
000005 //*
000006 //RUNIVP EXEC IGYWCLG,PARM.COBOL=RENT,REGION=1400K,
000007 //             PARM.LKED='LIST,XREF,LET,MAP'
000008 //COBOL.STEPLIB DD DSN=IGY.SIGYCOMP,
000009 //             DISP=SHR
000010 //COBOL.SYSIN DD *
000011        IDENTIFICATION DIVISION.
000012        PROGRAM-ID.    CALLIVP1.
000013        AUTHOR.        STUDENT PROGRAMMER.
000014        INSTALLATION.  MY UNIVERSITY
000015        DATE-WRITTEN.  JUL 27, 2004.
000016        DATE-COMPILED.
000017        /
000018        ENVIRONMENT DIVISION.
000019        CONFIGURATION SECTION.
000020        SOURCE-COMPUTER.  IBM-390.
000021        OBJECT-COMPUTER.  IBM-390.
000022
000023        PROCEDURE DIVISION.
000024            DISPLAY "***** HELLO WORLD *****" UPON CONSOLE.
000025            STOP RUN.
000026
000027 //GO.SYSOUT DD SYSOUT=*
000028 //
```

Figure 43. Procedure to compile, link-edit, and execute a COBOL program

During the execution of a step, the program is controlled by z/OS, not by JES (Figure 44 on page 87). Also, a spooling function is needed at this point in the process.

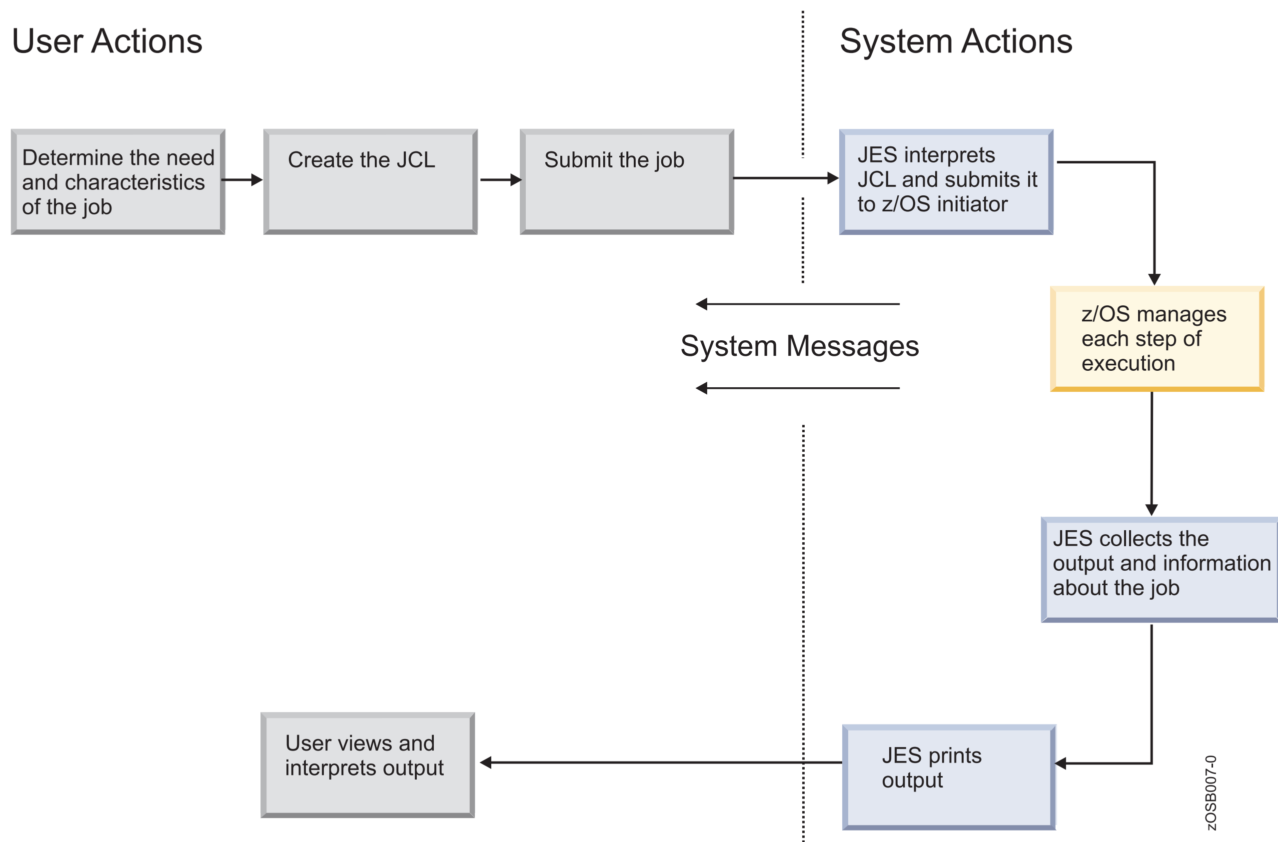


Figure 44. Related actions with JCL

Spooling is the means by which the system manipulates its work, including:

- Using storage on **direct access storage devices** (DASDs) as buffer storage to reduce processing delays when transferring data between peripheral equipment and a program to be run.
- Reading and writing input and output streams on an intermediate device for later processing or output.
- Performing an operation such as printing while the computer is busy with other work.

There are two sorts of spooling: Input and output. Both improve the performance of the program reading the input and writing the output.

To implement input spooling in JCL, you declare `// DD *`, which defines one file whose content records are in JCL between the `// DD *` statement and the `/*` statements. All the logical records must have 80 characters. In this case, this file is read and stored in a specific JES2 spool area (a huge JES file on disk) as shown in Figure 45 on page 88.

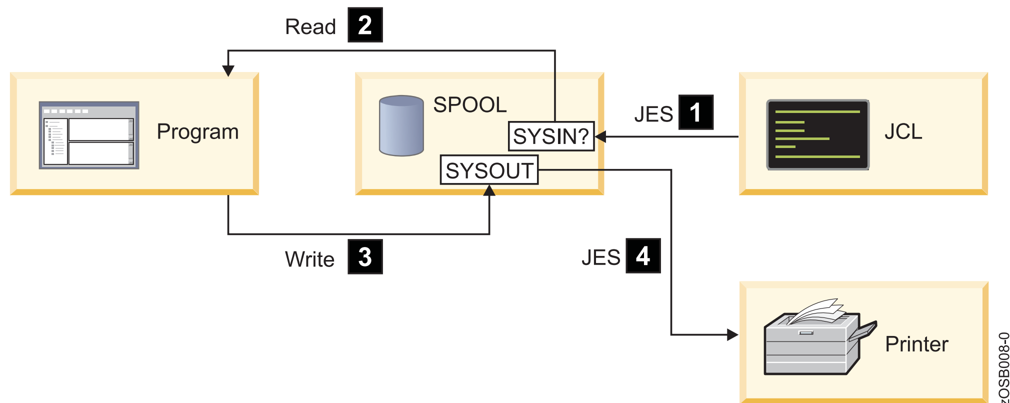


Figure 45. Spooling

Later, when the program is executed and asks to read this data, JES2 picks up the records in the spool and delivers them to the program (at disk speed).

To implement output spooling in JCL, you specify the keyword **SYSOUT** on the DD statement. **SYSOUT** defines an empty file in the spool, allocated with logical records of 132 characters in a printed format (EBCDIC/ASCII/UNICODE). This file is allocated by JES when interpreting a DD card with the **SYSOUT** keyword, and used later for the step program. Generally, after the end of the job, this file is printed by a JES function.

Batch processing and JES: Scenario 2

Suppose that you want to make a backup of one master file and then update the master file with records read in from another file (the **update** file).

If so, you need a job with two steps:

1. In Step 1, your job reads the master file and writes it to tape.
2. In Step 2, another program (which can be written in COBOL) is executed to read a record from the update file and searches for its match in the master file. The program updates the existing record (if it finds a match) or adds a new record if needed.

In this scenario, what kind of functions are needed in the operating system to meet your requirements? Your JCL must have two steps, the first one indicating the resources for the backup program, and the second for the update program.

- Who you are
- What resources are needed by the job, such as the following:
  - Load the backup program (that you already have compiled).
  - How much memory the system needs to allocate to accommodate the backup program, I/O buffers, and working areas.
  - Make accessible to the backup program an output tape data set to receive the backup, a copy, and the master file data set itself.
  - At program end indicate to the operating system that now your update program needs to be loaded into memory (however, this should not be done if the backup program failed).
  - Make accessible to the update program the update file and master file.
  - Make accessible to your program a printer for eventual messages.

Figure 46 illustrates the resources needed for each step.

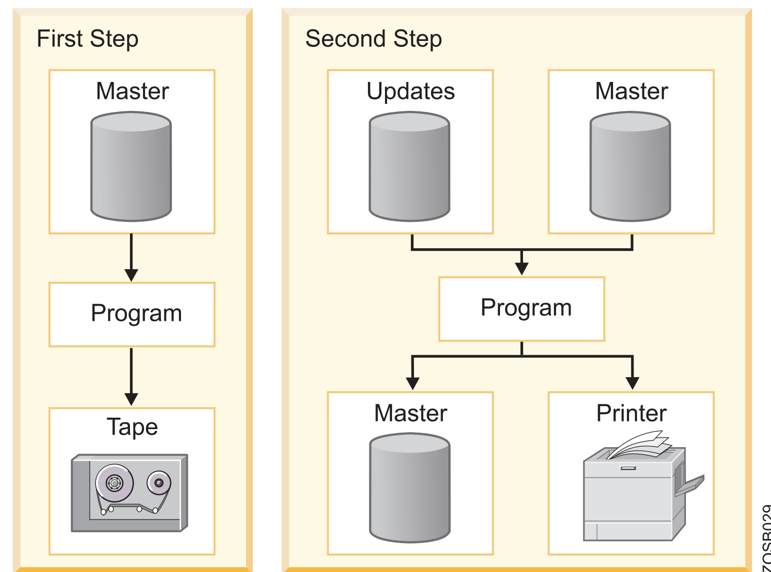


Figure 46. Scenario 2

Logically, the second step will not be executed if the first one fails for any reason. The second step will have a `// DD SYSOUT` statement to indicate the need for output spooling.

The jobs are allowed to start only when there are enough resources available. In this way, the system is made more efficient: JES manages jobs before and after running the program; the base control program manages jobs during processing.

Job flow through the system

During the life of a job, JES2 and the base control program of z/OS control different phases of the overall processing.

The job queues contain jobs that are waiting to run, currently running, waiting for their output to be produced, having their output produced, and waiting to be purged from the system.

Generally speaking, a job goes through the following phases:

1. Input
2. Conversion
3. Processing
4. Output
5. Print/punch (hard copy)
6. Purge

During batch job processing, numerous **checkpoints** occur. A checkpoint is a point in processing at which information about the status of a job and the system can be recorded (in a file called a checkpoint data set). Checkpoints allow the job step to be restarted later if it ends abnormally due to an error.

Figure 47 shows the different phases of a job during batch processing.

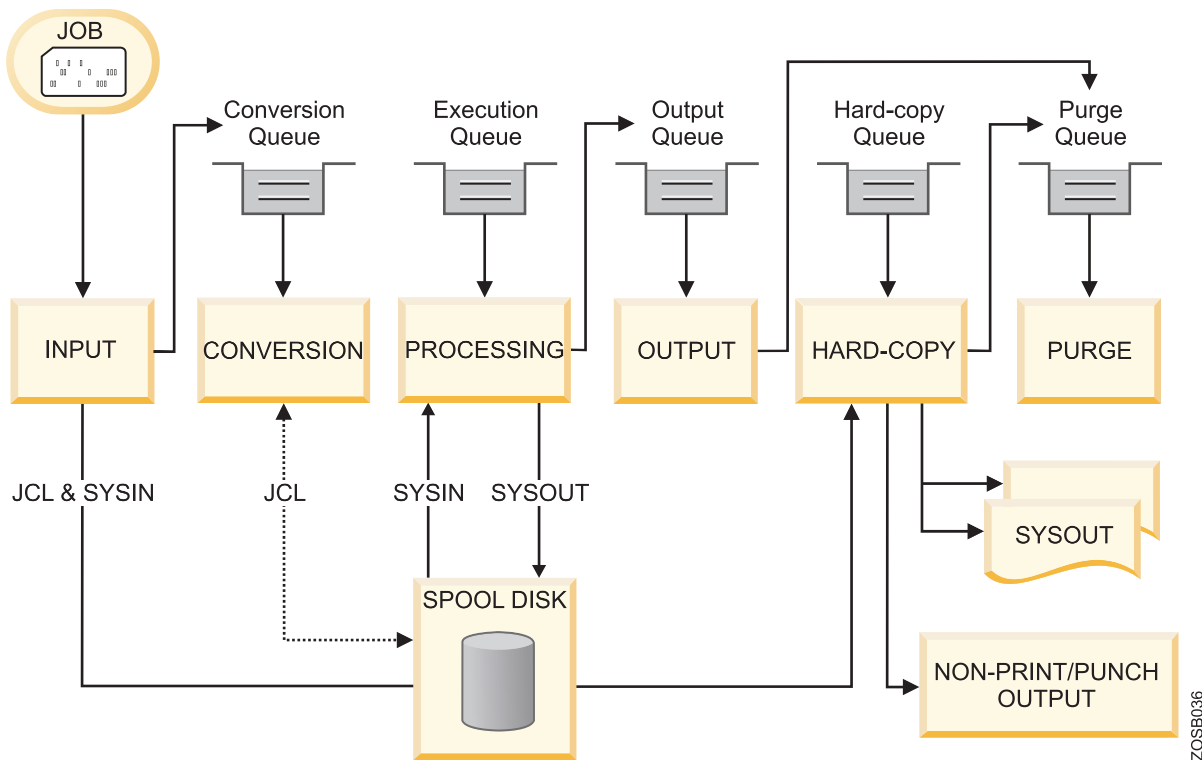


Figure 47. Job flow through the system

Input phase

JES2 accepts jobs, in the form of an input stream, from input devices, from other programs through internal readers, and from other nodes in a job entry network.

The internal reader is a program that other programs can use to submit jobs, control statements, and commands to JES2. Any job running in z/OS can use an internal reader to pass an input stream to JES2. JES2 can receive multiple jobs simultaneously through multiple internal readers. The system programmer defines internal readers to be used to process all batch jobs other than **started tasks** (STCs) and TSO requests.

JES2 reads the input stream and assigns a job identifier to each JOB JCL statement. JES2 places the job's JCL, optional JES2 control statements, and SYSIN data onto DASD data sets called spool data sets. JES2 then selects jobs from the spool data sets for processing and subsequent running.

Conversion phase

1. JES2 uses a converter program to analyze a job's JCL statements. The converter takes the job's JCL and merges it with JCL from a procedure library. The procedure library can be defined in the JCLLIB JCL statement, or system/user procedure libraries can be defined in the PROCxx DD statement of the JES2 startup procedure.
2. Then, JES2 converts the composite JCL into converter/interpreter text that both JES2 and the initiator can recognize.

3. Next, JES2 stores the converter/interpreter text on the spool data set. If JES2 detects any JCL errors, it issues messages, and the job is queued for output processing rather than execution. If there are no errors, JES2 queues the job for execution.

Processing phase

In the processing phase, JES2 responds to requests for jobs from the initiators. JES2 selects jobs that are waiting to run from a job queue and sends them to initiators.

An initiator is a system program belonging to z/OS, but controlled by JES or by the workload management (WLM) component of z/OS, which starts a job allocating the required resources to allow it to compete with other jobs that are already running.

JES2 initiators are initiators that are started by the operator or by JES2 automatically when the system initializes. They are defined to JES2 through JES2 initialization statements. To obtain an efficient use of available system resources, the installation associates each initiator with one or more job classes. Initiators select jobs whose classes match the initiator-assigned class, obeying the priority of the queued jobs.

WLM initiators are started by the system automatically based on performance goals, relative importance of the batch workload, and the capacity of the system to do more work. The initiators select jobs based on their service class and the order in which they were made available for execution. Jobs are routed to WLM initiators through a JOBCLASS JES2 initialization statement.

Output phase

JES2 controls all SYSOUT processing. SYSOUT is system-produced output; that is, all output produced by, or for, a job. This output includes system messages that must be printed, as well as data sets requested by the user that must be printed or punched. After a job finishes, JES2 analyzes the characteristics of the job's output in terms of its output class and device setup requirements; then JES2 groups data sets with similar characteristics. JES2 queues the output for print or punch processing.

Print/punch (hard copy) phase

JES2 selects output for processing from the output queues by output class, route code, priority, and other criteria. The output queue can have output that is to be processed locally or at a remote location. After processing all the output for a particular job, JES2 puts the job on the purge queue.

Purge phase

When all processing for a job completes, JES2 releases the spool space assigned to the job, making the space available for allocation to subsequent jobs. JES2 then issues a message to the operator indicating that the job has been purged from the system.

JES2 compared to JES3

IBM provides two kinds of job entry subsystems: JES2 and JES3. In many cases, JES2 and JES3 perform similar functions, but most installations use JES2.

Both JES2 and JES3 read jobs into the system, convert them to internal machine-readable form, select them for processing, process their output, and purge them from the system.

Some principle differences between the two JES systems include:

- In a mainframe installation that has only one processor, JES3 provides tape setup, dependent job control, and deadline scheduling for users of the system, while JES2 in the same system would require its users to manage these activities through other means. In an installation with a multiprocessor configuration, there are noticeable differences between the two, mainly in how JES2 exercises independent control over its job processing functions. That is, within the configuration, each JES2 processor controls its own job input, job scheduling, and job output processing.
- In cases where multiple z/OS systems are clustered (a **sysplex**), it is possible to configure JES2 to share spool and checkpoint data sets with other JES2 systems in the same sysplex. This configuration is called Multi-Access Spool (MAS). In contrast, JES3 exercises centralized control over its processing functions through a single global JES3 processor. This global processor provides all job selection, scheduling, and device allocation functions for all of the other JES3 systems.
- With JES3, installations may decide whether the global JES3 or z/OS base control program will handle device allocation. With JES2, only the z/OS base control program handles device allocation.

Chapter 5. Doing work on z/OS: How you submit, control and monitor jobs using JCL and SDSF

As a technical professional in the world of mainframe computing, you will need to know JCL, the language that tells z/OS which resources will be needed to process a batch job or to start a system task. You also will need to use SDSF to check the output of jobs and tasks submitted to the system.

Job control language (JCL) is used to tell the system what program to execute, followed by a description of program inputs and outputs. Basic JCL contains three types of statements: JOB, EXEC, and DD. A job can contain several EXEC statements (steps) and each step might have several DD statements. JCL provides a wide range of parameters and controls, but you will find that you use only a subset most of the time.

System users are expected to write simple JCL, but they normally use JCL procedures for more complex jobs. A cataloged procedure is written once and can then be used by many users. z/OS supplies many JCL procedures, and locally written ones can be added easily. To supply the parameters (usually DD statements) needed for a specific job, a user must understand how to override or extend statements in a JCL procedure.

After submitting a job, it is common to use **System Display and Search Facility** (SDSF), which is a utility that allows you to monitor, control, and view the output of jobs in the system.

What is JCL?

For every job that you submit, you need to tell z/OS where to find the appropriate input, how to process that input, and what to do with the resulting output. You use *job control language* (*JCL*) to convey this information to z/OS through a set of statements known as job control statements.

While application programmers need some knowledge of JCL, the production control analyst must be highly proficient with JCL, to create, monitor, correct and rerun the company's daily batch workload.

The set of job control statements is quite large, which allows you to provide a great deal of information to z/OS. Most jobs, however, can be run using a very small subset of these control statements. Once you become familiar with the characteristics of the jobs you typically run, you may find that you need to know the details of only some of the control statements.

The following JCL example represents a job that performs the same functions as the TSO commands outlined in Figure 20 on page 61.

```
//MYJOB     JOB 1
//MYSORT    EXEC PGM=SORT
//SORTIN    DD DISP=SHR,DSN=ZPROF.AREA.CODES
//SORTOUT   DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//SYSIN     DD *
SORT FIELDS=(1,3,CH,A)
/*
```

© Copyright IBM Corp. 2006, 2008

Each JCL DD statement is equivalent to the TSO ALLOCATE command. Both are used to associate a z/OS data set with a **ddname**, which is recognized by the program as an input or output. The difference in method of execution is that TSO executes the sort in the foreground while JCL is used to execute the sort in the background.

When submitted for execution:

MYJOB
Is a jobname the system associates with this workload.

MYSORT
Is the stepname, which instructs the system to execute the SORT program.

SORTIN
On the DD statement, SORTIN is the ddname. The SORTIN ddname is coded in the SORT program as a program input. The data set name (DSN) on this DD statement is **ZPROF**.AREA.CODES. The data set can be shared (DISP=SHR) with other system processes. The data content of **ZPROF**.AREA.CODES is SORT program input.

SORTOUT
This ddname is the SORT program output.

SYSOUT
SYSOUT=* specifies to send system output messages to the Job Entry Subsystem (JES) print output area. It is possible to send the output to a data set.

SYSIN
DD * is another input statement. It specifies that what follows is data or control statements. In this case, it is the sort instruction telling the SORT program which fields of the SORTIN data records are to be sorted.

"The Big Three" JCL statements: JOB, EXEC, and DD

All jobs require the three main types of JCL statements: JOB, EXEC, and DD. A *job* defines a specific workload for z/OS to process. A *job* is a separately executable unit of work defined by a user, and run by a computer. This representation of a unit of work consists of one program or a set of programs, files, and control statements.

Some z/OS users use the older term *JCL card* instead of JCL statement, because JCL used to be submitted to the system in the form of punched cards. Now JCL resides in storage (data sets) rather than on punched cards.

Because JCL was originally designed for punched cards, the details of coding JCL statements can be complicated. However, the general concepts are quite simple, and most jobs can be run using a very small subset of these control statements.

JCL has three basic statements:

JOB Labels the unit of work that you want the system to perform, by providing a name (jobname). The JOB statement can optionally include accounting information and parameters that apply to the entire job.

A *job stream*, or input stream, consists of one or more jobs that are submitted to the system in a sequence. You can submit jobs to z/OS through either TSO or ISPF.

EXEC Provides the name of an application program or JCL procedure (sometimes called a “proc”) that the system is to run (or execute). A single job may contain multiple EXEC statements. Each EXEC statement within the same job is a *job step*.

DD Identifies input and output to the program or procedure on the EXEC statement. Each DD (data definition) statement links a data set or other I/O device or function to a name (ddname) coded in the program. DD statements are associated with a particular job step.

Two special DD statements, JOBLIB DD and STEPLIB DD, identify the location of the program or procedure on the EXEC statement. z/OS automatically searches standard system libraries, so you need to code these special DD statements in your JCL only when your program or procedure resides in a private library.

How is a job submitted for batch processing?

You can use several methods of submitting a job for batch processing; most involve either TSO or ISPF.

Using UNIX as an analogy, a UNIX process can be processed in the background by appending an ampersand (&) to the end of a command or script. Pressing Enter then submits the work as a background process.

In z/OS terminology, work (a job) is submitted for batch processing. Batch processing is a rough equivalent to UNIX background processing. The job runs independently of the interactive session. The term batch is used because it is a large collection of jobs that can be queued, waiting their turn to be executed when the needed resources are available. Commands to submit jobs might take any of the following forms:

ISPF editor command line
Type `submit` or `sub` and press Enter.

ISPF command shell
When your JCL data set is sequential, type `submit `USER.JCL'` and press Enter.

ISPF command line
- When your JCL data set is sequential, type `TSO submit 'USER.JCL'` and press Enter.
- When your JCL data set is a library or partitioned data set containing the member you want to submit, type `TSO submit 'USER.JCL(MYJOB)'` and press Enter.

TSO command line
Type `submit 'USER.JCL'` and press Enter.

What is the System Display and Search Facility (SDSF)?

System Display and Search Facility (SDSF) is a utility that allows you to monitor, control, and view the output of jobs in the system.

After submitting a job, it is common to use **System Display and Search Facility** (SDSF) to review the output for successful completion or to review and correct JCL errors. SDSF allows you to display printed output held in the JES spool area. Much

of the printed output sent to JES by batch jobs (and other jobs) is never actually printed. Instead it is inspected using SDSF and deleted or used as needed.

SDSF provides a number of additional functions, including:
- Viewing the system log and searching for any literal string
- Entering system commands (in earlier versions of the operating system, only the operator could enter commands)
- Controlling job processing (hold, release, cancel, and purge jobs)
- Monitoring jobs while they are being processed
- Displaying job output before deciding to print it
- Controlling the order in which jobs are processed
- Controlling the order in which output is printed
- Controlling printers and initiators

Figure 48 shows the SDSF primary option menu.

```
   Display  Filter  View  Print  Options  Help
------------------------------------------------------------------------------
ISPFCU41                          SDSF Primary Option Menu
COMMAND INPUT  ===> _                                        SCROLL ===> PAGE

DA    Active users                               INIT  Initiators
I     Input queue                                PR    Printers
O     Output queue                               PUN   Punches
H     Held output queue                          RDR   Readers
ST    Status of jobs                             LINE  Lines
                                                 NODE  Nodes
LOG   System log                                 SO    Spool offload
SR    System requests                            SP    Spool volumes
MAS   Members in the MAS
JC    Job classes                                ULOG  User session log
SE    Scheduling environments
RES   WLM resources
ENC   Enclaves
PS    Processes

END   Exit SDSF

Licensed Materials - Property of IBM

5694-A01 (C) Copyright IBM Corp. 1981, 2002. All rights reserved.
US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

 F1=HELP      F2=SPLIT     F3=EXIT      F4=RETURN    F5=IFIND     F6=BOOK
 F7=UP        F8=DOWN      F9=SWAP      F10=LEFT    F11=RIGHT    F12=RETRIEVE
```

Figure 48. SDSF primary option menu

SDSF uses a hierarchy of online panels to guide users through its functions, as shown in Figure 49 on page 97.

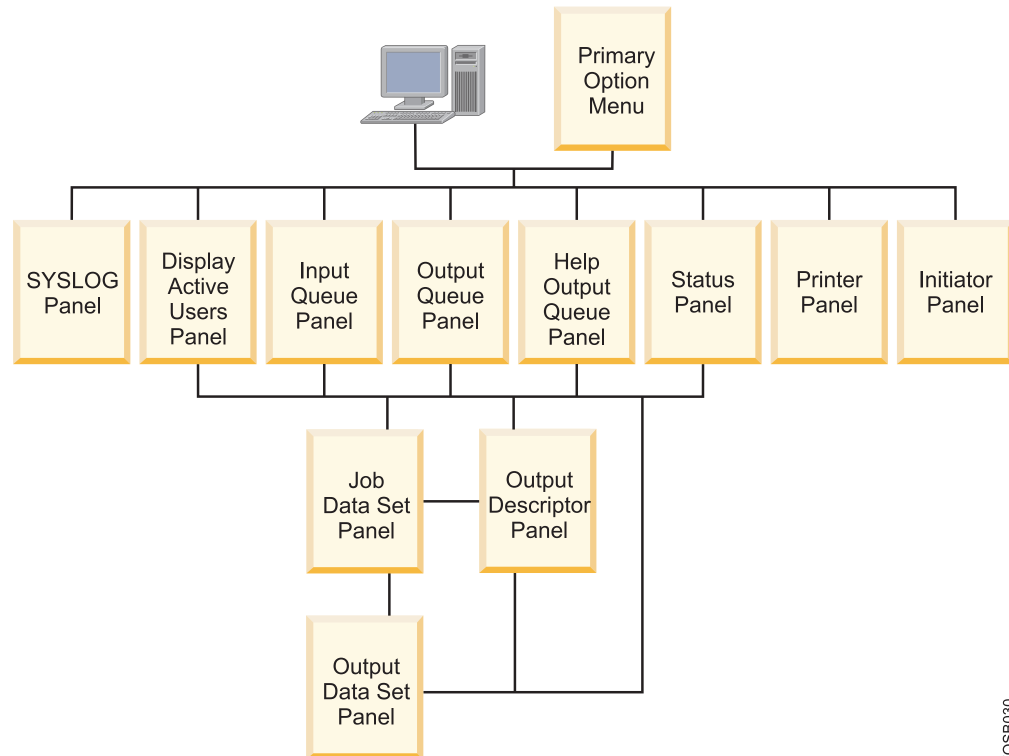


Figure 49. SDSF panel hierarchy

You can see the JES output data sets created during the execution of your batch job. They are saved on the JES spool data set.

You can see the JES data sets in any of the following queues:

I Input queue
DA Execution queue
O Output queue
H Held queue
ST Status queue

For output and held queues, you cannot see those JES data sets that you requested to be automatically purged (by setting a MSGCLASS sysout class that has been defined to not save output). Also, depending on the MSGCLASS you chose on the JOB card, the sysouts can be either in the output queue or in the held queue.

```
Screen 1

   Display  Filter  View  Print  Options  Help
-------------------------------------------------------------------------------
SDSF HELD OUTPUT DISPLAY ALL CLASSES   LINES 44
COMMAND INPUT  ===> _                                   SCROLL ===> PAGE
PREFIX=*  DEST=(ALL)   OWNER=*  SYSNAME=
NP   JOBNAME   JobID    Owner    Prty C ODisp  Dest              Tot-Rec  Tot-
?_   MIRIAM2   JOB26044 MIRIAM    144 T HOLD   LOCAL                  44
```

```
Screen 2

   Display  Filter  View  Print  Options  Help
-------------------------------------------------------------------------------
SDSF JOB DATA SET  DISPLAY  - JOB MIRIAM2 (JOB26044)   LINE 1-3 (3)
COMMAND INPUT  ===> _                                   SCROLL ===> PAGE
PREFIX=*  DEST=(ALL)   OWNER=*  SYSNAME=
NP   DDNAME   StepName  ProcStep DSID  Owner   C Dest             Rec-Cnt  Page
     JESMSGLG JES2                  2  MIRIAM  T LOCAL                 20
     JESJCL   JES2                  3  MIRIAM  T LOCAL                 12
     JESYSMSG JES2                  4  MIRIAM  T LOCAL                 12
```

Figure 50. SDSF viewing the JES2 Output files

Screen 1 in Figure 50 displays a list of jobs that submitted, with output directed to the held (Class T) queue, as identified in the **MSGCLASS=T** parameter on the job card. In this case, only one job has been submitted and executed. Therefore, only one job is on the held queue. To display the output files generated by job 7359, you would enter a ? command in the NP column.

Screen 2 in Figure 50 displays three ddnames: the JES2 messages log file, the JES2 JCL file, and the JES2 system messages file. This option is useful when you are seeing jobs with many files directed to SYSOUT and you want to display one associated with a specific step. To see all output files, you would enter an S in the NP column. The JES2 job log is displayed similar to the one shown in Figure 51 on page 99.

```
J E S 2  J O B  L O G  --  S Y S T E M  S C 6 4  --  N O D E

13.19.24 JOB26044 ---- WEDNESDAY, 27 AUG 2003 ----
13.19.24 JOB26044  IRR010I  USERID MIRIAM   IS ASSIGNED TO THIS JOB.
13.19.24 JOB26044  ICH70001I MIRIAM   LAST ACCESS AT 13:18:53 ON WEDNESDAY,
AUGUST 2003
13.19.24 JOB26044  $HASP373 MIRIAM2  STARTED - INIT 1    - CLASS A- SYS SC64
13.19.24 JOB26044  IEF403I MIRIAM2 - STARTED - ASID=0027 - SC64
13.19.24 JOB26044  -                                        --TIMINGS (MINS.)--
13.19.24 JOB26044  -JOBNAME  STEPNAME PROCSTEP    RC   EXCP    CPU  SRB  CLOCK
13.19.24 JOB26044  -MIRIAM2           STEP1       00      9    .00  .00  .00
13.19.24 JOB26044  IEF404I MIRIAM2 - ENDED - ASID=0027 - SC64
13.19.24 JOB26044  -MIRIAM2  ENDED.  NAME-MIRIAM
TOTAL CPU TIME=
13.19.24 JOB26044  $HASP395 MIRIAM2  ENDED
------ JES2 JOB STATISTICS ------
27 AUG 2003 JOB EXECUTION DATE
11 CARDS READ
44 SYSOUT PRINT RECORDS
0 SYSOUT PUNCH RECORDS
3 SYSOUT SPOOL KBYTES
0.00 MINUTES EXECUTION TIME
1 //MIRIAM2 JOB 19,MIRIAM,NOTIFY=&SYSUID,MSGCLASS=T,
  // MSGLEVEL=(1,1),CLASS=A
  IEFC653I SUBSTITUTION JCL -
  19,MIRIAM,NOTIFY=MIRIAM,MSGCLASS=T,MSGLEVE
2 //STEP1 EXEC PGM=IEFBR14
  //*-------------------------------------------------*
  //* THIS IS AN EXAMPLE OF A NEW DATA SET ALLOCATION
  //*-------------------------------------------------*
3 //NEWDD DD   DSN=MIRIAM.IEFBR14.TEST1.NEWDD,
  //          DISP=(NEW,CATLG,DELETE),UNIT=SYSDA,
  //          SPACE=(CYL,(10,10,45)),LRECL=80,BLKSIZE=3120
4 //SYSPRINT  DD  SYSOUT=T
  /*
ICH70001I MIRIAM   LAST ACCESS AT 13:18:53 ON WEDNESDAY, AUGUST 27, 2003
IEF236I ALLOC. FOR MIRIAM2 STEP1
IGD100I 390D ALLOCATED TO DDNAME NEWDD    DATACLAS (        )
IEF237I JES2 ALLOCATED TO SYSPRINT
IEF142I MIRIAM2 STEP1 - STEP WAS EXECUTED - COND CODE 0000
IEF285I   MIRIAM.IEFBR14.TEST1.NEWDD                   CATALOGED
IEF285I   VOL SER NOS= SBOX38.
IEF285I   MIRIAM.MIRIAM2.JOB26044.D0000101.?           SYSOUT
IEF373I STEP/STEP1   /START 2003239.1319
IEF374I STEP/STEP1   /STOP  2003239.1319 CPU    0MIN 00.00SEC SRB 0MIN 00.00S
IEF375I  JOB/MIRIAM2 /START 2003239.1319
IEF376I  JOB/MIRIAM2 /STOP  2003239.1319 CPU    0MIN 00.00SEC SRB 0MIN 00.00S
```

Figure 51. Sample JES2 job log

Chapter 6. Parallel Sysplex: Worth the effort for continuous availability

Parallel Sysplex technology is an enabling technology, allowing highly reliable, redundant, and robust mainframe technologies to achieve near-continuous availability.

A properly configured Parallel Sysplex cluster is designed to remain available to its users and applications with minimal downtime, for example:

- Hardware and software components provide for concurrency to facilitate non-disruptive maintenance, like Capacity Upgrade on Demand, which allows processing or coupling capacity to be added one engine at a time without disruption to running workloads.
- DASD subsystems employ disk mirroring or RAID technologies to help protect against data loss, and exploit technologies to enable point-in-time backup, without the need to shut down applications.
- Networking technologies deliver functions such as VTAM® Generic Resources, Multi-Node Persistent Sessions, Virtual IP Addressing, and Sysplex Distributor to provide fault-tolerant network connections.
- I/O subsystems support multiple I/O paths and dynamic switching to prevent loss of data access and improved throughput.
- z/OS software components allow new software releases to coexist with lower levels of those software components to facilitate rolling maintenance.
- Business applications are ″data sharing-enabled″ and cloned across servers to allow workload balancing to prevent loss of application availability in the event of an outage.
- Operational and recovery processes are fully automated and transparent to users, and reduce or eliminate the need for human intervention.

Benefits of Parallel Sysplex: No single points of failure

In a Parallel Sysplex cluster, it is possible to construct a parallel processing environment with no single points of failure. Because all of the systems in the Parallel Sysplex can have concurrent access to all critical applications and data, the loss of a system due to either hardware or software failure does not necessitate loss of application availability.

Peer instances of a failing subsystem executing on remaining healthy system nodes can take over recovery responsibility for resources held by the failing instance. Alternatively, the failing subsystem can be automatically restarted on still-healthy systems using automatic restart capabilities to perform recovery for work in progress at the time of the failure. While the failing subsystem instance is unavailable, new work requests can be redirected to other data-sharing instances of the subsystem on other cluster nodes to provide continuous application availability across the failure and subsequent recovery. This alternatives provide the ability to mask planned as well as unplanned outages to the end user.

Because of the redundancy in the configuration, there is a significant reduction in the number of single points of failure. Without a Parallel Sysplex, the loss of a server could severely impact the performance of an application, as well as introduce system management difficulties in redistributing the workload or

© Copyright IBM Corp. 2006, 2008

reallocating resources until the failure is repaired. In a Parallel Sysplex environment, it is possible that the loss of a server may be transparent to the application, and the server workload can be redistributed automatically within the Parallel Sysplex with little performance degradation. In a Parallel Sysplex environment, events that otherwise would seriously impact application availability, such as failures in central processor complex (CPC) hardware elements or critical operating system components, have reduced impact.

Even though they work together and present a single image, the nodes in a Parallel Sysplex cluster remain individual systems, making installation, operation, and maintenance non-disruptive. The system programmer can introduce changes, such as software upgrades, one system at a time, while the remaining systems continue to process work. This design allows the mainframe IT staff to roll changes through its systems on a schedule that is convenient to the business.

Benefits of Parallel Sysplex: Capacity and scaling

The Parallel Sysplex environment can scale nearly linearly from 2 to 32 systems. These systems can be a mix of any servers that support the Parallel Sysplex environment. The aggregate capacity of this configuration meets every processing requirement known today.

Benefits of Parallel Sysplex: Dynamic workload balancing

The entire Parallel Sysplex cluster can be viewed as a single logical resource to end users and business applications. Just as work can be dynamically distributed across the individual processors within a single SMP server, so too, can work be directed to any node in a Parallel Sysplex cluster having available capacity. This capability avoids the need to partition data or applications among individual nodes in the cluster or to replicate databases across multiple servers.

Workload balancing also permits a business to run diverse applications across a Parallel Sysplex cluster while maintaining the response levels critical to a business. The mainframe IT director selects the service level agreements required for each workload, and the workload management (WLM) component of z/OS, along with subsystems such as CP/SM or IMS, automatically balances tasks across all the resources of the Parallel Sysplex cluster to meet these business goals. The work can come from a variety of sources, such as batch, SNA, TCP/IP, DRDA®, or WebSphere MQ.

There are several aspects to consider for recovery:

1. First, when a failure occurs, it is important to bypass it by automatically redistributing the workload to utilize the remaining available resources.

   After the failing element has been isolated, it is necessary to non-disruptively redirect the workload to the remaining available resources in the Parallel Sysplex. In the event of failure in the Parallel Sysplex environment, the online transaction workload is automatically redistributed without operator intervention.
2. Secondly, it is necessary to recover the elements of work that were in progress at the time of the failure.
3. Finally, when the failed element is repaired, it should be brought back into the configuration as quickly and transparently as possible to again start processing the workload. Parallel Sysplex technology enables all this to happen.

Generic resource management provides the ability to specify to VTAM a common network interface. This can be used for CICS terminal owning regions (TORs), IMS Transaction Manager, TSO, or DB2 DDF work. If one of the CICS TORs fails, for example, only a subset of the network is affected. The affected terminals are able to immediately log on again and continue processing after being connected to a different TOR.

Benefits of Parallel Sysplex: Ease of use

The Parallel Sysplex solution satisfies a major customer requirement for continuous 24-hour-a-day, 7-day-a-week availability, while providing techniques for achieving simplified Systems Management consistent with this requirement. Some of the features of the Parallel Sysplex solution that contribute to increased availability also help to eliminate some Systems Management tasks.

Workload management (WLM) component
The workload management (WLM) component of z/OS provides sysplex-wide workload management capabilities based on installation-specified performance goals and the business importance of the workloads. WLM tries to attain the performance goals through dynamic resource distribution. WLM provides the Parallel Sysplex cluster with the intelligence to determine where work needs to be processed and in what priority. The priority is based on the customer's business goals and is managed by sysplex technology.

Sysplex Failure Manager (SFM)
The Sysplex Failure Management policy allows the installation to specify failure detection intervals and recovery actions to be initiated in the event of the failure of a system in the sysplex.

Without SFM, when one of the systems in the Parallel Sysplex fails, the operator is notified and prompted to take some recovery action. The operator may choose to partition the non-responding system from the Parallel Sysplex, or to take some action to try to recover the system. This period of operator intervention might tie up critical system resources required by the remaining active systems. Sysplex Failure Manager allows the installation to code a policy to define the recovery actions to be initiated when specific types of problems are detected, such as fencing off the failed image that prevents access to shared resources, logical partition deactivation, or central storage and expanded storage acquisition, to be automatically initiated following detection of a Parallel Sysplex failure.

Automatic Restart Manager (ARM)
Automatic Restart Manager enables fast recovery of subsystems that might hold critical resources at the time of failure. If other instances of the subsystem in the Parallel Sysplex need any of these critical resources, fast recovery will make these resources available more quickly. Even though automation packages are used today to restart the subsystem to resolve such deadlocks, ARM can be activated closer to the time of failure.

ARM reduces operator intervention in the following areas:

- Detection of the failure of a critical job or started task
- Automatic restart after a started task or job failure

  After an abend of a job or started task, the job or started task can be restarted with specific conditions, such as overriding the original JCL or specifying job dependencies, without relying on the operator.

- Automatic redistribution of work to an appropriate system following a system failure

  This removes the time-consuming step of human evaluation of the most appropriate target system for restarting work

Cloning and symbolics
: **Cloning** refers to replicating the hardware and software configurations across the different physical servers in the Parallel Sysplex. That is, an application that is going to take advantage of parallel processing might have identical instances running on all images in the Parallel Sysplex. The hardware and software supporting these applications could also be configured identically on all systems in the Parallel Sysplex to reduce the amount of work required to define and support the environment.

  The concept of **symmetry** allows new systems to be introduced and enables automatic workload distribution in the event of failure or when an individual system is scheduled for maintenance. It also reduces the amount of work required by the system programmer in setting up the environment. Note that symmetry does **not** preclude the need for systems to have unique configuration requirements, such as the asymmetric attachment of printers and communications controllers, or asymmetric workloads that do not lend themselves to the parallel environment.

  System symbolics are used to help manage cloning. z/OS provides support for the substitution values in startup parameters, JCL, system commands, and started tasks. These values can be used in parameter and procedure specifications to allow unique substitution when dynamically forming a resource name.

zSeries resource sharing
: A number of base z/OS components have discovered that the IBM coupling facility shared storage provides a medium for sharing component information for the purpose of multisystem resource management. This exploitation, called IBM zSeries Resource Sharing, enables sharing of physical resources such as files, tape drives, consoles, and catalogs with improvements in cost, performance and simplified systems management. This is **not to be confused** with Parallel Sysplex data sharing by the database subsystems. zSeries Resource Sharing delivers immediate value even for customers who are not leveraging data sharing, through native system exploitation delivered with the base z/OS software stack.

  One of the goals of the Parallel Sysplex solution is to provide simplified systems management by reducing complexity in managing, operating, and servicing a Parallel Sysplex, without requiring an increase in the number of support staff and without reducing availability.

Benefits of Parallel Sysplex: Single system image

Even though there could be multiple servers and z/OS images in the Parallel Sysplex and a mix of different technologies, the collection of systems in the Parallel Sysplex should appear as a single entity to the operator, the end user, the database administrator, and so on. A single system image brings reduced complexity from both operational and definition perspectives.

Regardless of the number of system images and the complexity of the underlying hardware, the Parallel Sysplex solution provides for a single system image from several perspectives:

- Data access, allowing dynamic workload balancing and improved availability

- Dynamic Transaction Routing, providing dynamic workload balancing and improved availability
- End-user interface, allowing logon to a logical network entity
- Operational interfaces, allowing easier Systems Management

It is a requirement that the collection of systems in the Parallel Sysplex can be managed from a logical single point of control. The term ″single point of control″ means the ability to access whatever interfaces are required for the task in question, without reliance on a physical piece of hardware. For example, in a Parallel Sysplex of many systems, it is necessary to be able to direct commands or operations to any system in the Parallel Sysplex, without the necessity for a console or control point to be physically attached to every system in the Parallel Sysplex.

Even though individual hardware elements or entire systems in the Parallel Sysplex fail, a single system image must be maintained. This means that, as with the concept of single point of control, the presentation of the single system image is not dependent on a specific physical element in the configuration. From the end-user point of view, the parallel nature of applications in the Parallel Sysplex environment must be transparent. An application should be accessible regardless of which physical z/OS image supports it.

Benefits of Parallel Sysplex: Compatible change and non-disruptive growth

A primary goal of Parallel Sysplex is continuous availability. Therefore, it is a requirement that changes such as new applications, software, or hardware can be introduced non-disruptively, and that they be able to coexist with current levels. In support of compatible change, the hardware and software components of the Parallel Sysplex solution will allow the coexistence of two levels, that is, level N and level N+1. This means, for example, that no IBM software product will make a change that cannot be tolerated by the previous release.

Benefits of Parallel Sysplex: Application compatibility

A design goal of Parallel Sysplex clustering is that no application changes be required to take advantage of the technology. For the most part, this has held true, although some affinities need to be investigated to get the maximum advantage from the configuration.

From the application architects' point of view, three major points might lead to the decision to run an application in a Parallel Sysplex:

Technology benefits

Scalability (even with non-disruptive upgrades), availability, and dynamic workload management are tools that enable an architect to meet customer needs in cases where the application plays a key role in the customer's business process. With the multisystem data sharing technology, all processing nodes in a Parallel Sysplex have full concurrent read/write access to shared data without affecting integrity and performance.

Integration benefits

Because many applications are historically S/390- and z/OS-based, new applications on z/OS get performance and maintenance benefits, especially if they are connected to existing applications.

Infrastructure benefits

If there is already an existing Parallel Sysplex, it needs very little infrastructure work to integrate a new application. In many cases the installation does not need to integrate new servers. Instead it can leverage the existing infrastructure and make use of the strengths of the existing sysplex. With Geographically Dispersed Parallel Sysplex™ (GDPS®)— connecting multiple sysplexes in different locations— the mainframe IT staff can create a configuration that is enabled for disaster recovery.

Benefits of Parallel Sysplex: Disaster recovery

Geographically Dispersed Parallel Sysplex (GDPS) is the primary disaster recovery and continuous availability solution for a mainframe-based multi-site enterprise.

GDPS automatically mirrors critical data and efficiently balances workload between the sites. GDPS also uses automation and Parallel Sysplex technology to help manage multi-site databases, processors, network resources and storage subsystem mirroring. This technology offers continuous availability, efficient movement of workload between sites, resource management, and prompt data recovery for business-critical mainframe applications and data. With GDPS, the current maximum distance between the two sites is 100km (about 62 miles) of fiber, although there are some other restrictions. This provides a synchronous solution that helps to ensure no loss of data.

There is also GDPS/XRC, which can be used over extended distances and should provide a recovery point objective of less than two minutes (that is, a maximum of two minutes of data would need to be recovered or is lost).

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corp. 2006, 2008

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This book documents information that is NOT intended to be used as Programming Interfaces of z/OS.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at ″Copyright and trademark information″ at http://www.ibm.com/legal/copytrade.shtml

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft®, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Printed in USA