

GA21-9330-4

File No. S38-01

IBM System/38

IBM System/38
Functional Concepts Manual



GA21-9330-4

File No. S38-01

IBM System/38

IBM System/38 Functional Concepts Manual



Fifth Edition (September 1985)

This major revision makes obsolete GA21-9330-3. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition. See *About This Manual* for a summary of changes.



The functions described in this publication apply to the IBM System/38 machine interface. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include 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.

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

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

This publication could contain technical inaccuracies or typographical errors. A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department 245, Rochester Minnesota, U.S.A. 55901. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.



Contents

ABOUT THIS MANUAL	vii	CONTEXT MANAGEMENT	2-44
Summary of Changes	vii	Kinds of Contexts	2-44
What You Should Know	viii	Machine Context	2-44
If You Need More Information	viii	User-Defined Context	2-44
Communications	viii	Context Management Functions	2-44
Device Operation	viii	Materializing Contexts	2-45
Problem Handling	viii	Context Authorizations	2-45
Machine Interface Programming Information	viii	AUTHORIZATION MANAGEMENT	2-46
CHAPTER 1. INTRODUCTION	1-1	User Profiles	2-46
Object Concepts	1-1	Adopted User Profile	2-47
Integrity and Authorization	1-4	Object Authorization	2-48
CHAPTER 2. OBJECT CONTROL FUNCTIONS	2-1	Special Authorizations	2-55
OBJECT MANAGEMENT	2-1	Resource Authorization	2-56
System Objects	2-1	Privileged Instructions	2-56
System Object Descriptions	2-1	Authorization Functions	2-56
System Object Characteristics	2-6	Enrolling Users	2-56
Common Attributes of System Objects	2-8	Modifying Authorization	2-57
Program Objects	2-9	Materializing Authority	2-58
Data Objects	2-10	Authority Verification	2-58
Constant Data Objects	2-22	CHAPTER 3. PROGRAM FUNCTIONS	3-1
Entry Point	2-22	PROGRAM MANAGEMENT	3-1
Branch Point	2-22	Program Creation	3-1
Instruction Definition List	2-23	Instruction Stream	3-1
Operand List	2-23	Object Definition Table	3-2
Exception Description	2-23	User Data	3-3
Space Pointer Machine Object	2-24	Program Optimization	3-3
ADDRESSING	2-26	Program Destruction	3-4
ODT Addressing	2-26	Program Materialization	3-4
Pointer Addressing	2-26	Object Mapping Table	3-4
Pointer Data Objects	2-26	Program Observability Deletion	3-4
Space Pointer Machine Object	2-27	COMPUTATION AND BRANCHING	3-5
Space Pointer	2-27	Computational and Branching Capabilities	3-5
Data Pointer	2-29	Computational Operands	3-5
System Pointer	2-29	Computational Characteristics	3-5
Instruction Pointer	2-29	Computational Instructions and Data Descriptions	3-5
System Object Addressing	2-30	Generic Computational Operations	3-6
Symbolic Address	2-30	Attribute Binding	3-7
System Pointer Addressing	2-30	Operand Overlap	3-7
System Object Address Resolution	2-30	Avoiding Invalid Results	3-11
Space Addressing	2-32	Optional Computational Instruction Forms	3-12
Data Object Addressing	2-32	Arithmetic Operations	3-14
Data Object Address Resolution	2-35	Binary Computation	3-14
Array Addressing	2-37	Packed Decimal Computation	3-14
Substring Addressing	2-37	Zoned Decimal Computation	3-14
Space Extent Checking	2-38	Floating-Point Computation	3-15
Argument and Parameter Addressing	2-39	Floating-Point Overflow	3-21
Arguments	2-39	Floating-Point Underflow	3-22
Parameters	2-39	Floating-Point Zero Divide	3-23
Argument Lists	2-40	Floating-Point Inexact Result	3-23
Parameter Lists	2-41	Floating-Point Invalid Operand	3-24
Argument/Parameter Correspondence	2-41	Arithmetic Instructions	3-25
Process Addressing	2-43	Character String Operations	3-25
Instruction Addressing	2-43	Character String Instructions	3-25
Instruction Numbers	2-43	Boolean Operations	3-27
Instruction Pointers	2-43	Boolean Instructions	3-27
Instruction Definition Lists	2-43	Comparison Operations	3-28
		Comparison Instructions	3-28

Object Movement and Conversion Operations	3-29	Event Signaling	4-18
Movement Instructions	3-29	Machine Event Signaling	4-18
Conversion Instructions	3-30	Signaling by Signal Event Instruction	4-18
Branching Operations	3-33	Signaling an Event to a Process	4-19
Unconditional Branching	3-33	Conditions for Signaling an Event Monitor	4-19
Conditional Branching	3-34	Event Handling	4-19
Variable Branching	3-34	Asynchronous Event Handling	4-19
Editing Operations	3-34	Synchronous Event Handling	4-20
Editing Instructions	3-35	Event-Related Data	4-21
Logical Character Operations	3-35	Event Rules	4-22
Array Index Operations	3-35	EXCEPTION MANAGEMENT	4-23
No Operation	3-35	Exception Descriptions	4-23
PROGRAM EXECUTION	3-36	Exception Detection and Signaling	4-25
Program Activation	3-36	Locating an Exception Description	4-25
Activation Creation	3-36	Exception Handling	4-26
Activation Destruction	3-38	Ignored Exceptions	4-26
Program Invocation	3-38	Deferred Exception Handling	4-27
Invocation Creation	3-38	Immediate Exception Handling	4-27
Invocation Destruction	3-42	Retrieving Exception-Related Data Option	4-28
Invocation Example	3-45	Returns from Exception Handling	4-29
Subinvocations	3-47	Exception-Related Data	4-31
Arguments and Parameters	3-48	Exception-Related Data Option	4-31
Interinvocation Communications	3-49	RESOURCE MANAGEMENT	4-32
Intrainvocation Communications	3-53	Resources	4-33
Interprocess Communications	3-53	Processor Resource	4-33
CHAPTER 4. SUPERVISOR AND CONTROL		Storage Resource	4-33
FUNCTIONS	4-1	System Objects	4-34
PROCESS MANAGEMENT	4-1	Control and Monitoring Functions	4-35
Establishing a Process	4-1	Multiprogramming Level Control	4-35
Process Control Space	4-1	Storage Resource Functions	4-37
Process Definition Template	4-1	Process Attributes	4-40
Process Structure	4-1	System Object Locks	4-41
Process Initiation Steps	4-2	Types of System Objects That Can Be Locked	4-41
Process Domain	4-2	Lock Request Granting Algorithm	4-41
Process States	4-2	Sharing Data Within a System Object	4-44
Process Phases	4-4	Implicit Locks	4-44
Sequencing through Process Phases	4-5	Transferring Locks	4-44
Process Authorization	4-5	Locking a Space Location	4-44
Object Address Resolution	4-6	Unlocking a Space Location	4-44
External Data Object Resolution	4-6	Materializing Locks	4-44
Process Management Instructions	4-7	Unlocking System Objects	4-45
Process Control Space Instructions	4-7	Deadlock	4-45
Process Control Instructions	4-7	Deadlock Detection and Resolution	4-47
Authority for Process Control Instruction Usage	4-8	CHAPTER 5. DATA FUNCTIONS	5-1
Process Attributes	4-9	DATA BASE MANAGEMENT	5-1
Process Control Attributes	4-9	Major Data Base Objectives and Characteristics	5-1
Resource Management Attributes	4-10	Data Base Objects	5-2
Process Pointer Attributes	4-10	Using Data Base Functions	5-3
Process Status Indicators	4-11	Creating a Data Space	5-3
Process Resource Usage Attributes	4-12	Creating a Cursor	5-3
Subordinate Processes Identification	4-12	Activating a Cursor	5-4
Process Performance Attributes	4-12	Inserting an Entry	5-4
Interprocess Communication	4-12	Finding an Entry without a Data Space Index	5-4
Object Locks	4-13	Retrieving an Entry	5-4
Process Exception Handling	4-14	Updating an Entry	5-5
Process Control Instruction Characteristics	4-14	Deleting an Entry	5-5
EVENT MANAGEMENT	4-15	De-activating a Cursor	5-5
Events	4-15	Destroying a Cursor	5-5
Event Identification	4-15	Destroying a Data Space	5-5
Event Monitoring	4-16	Creating a Data Space Index	5-5
		Finding an Entry with a Data Space Index	5-6
		Destroying a Data Space Index	5-6
		Copying Data Space Entries	5-6
		Shared Data Spaces	5-7

Multiple Locked Entries	5-8	System to System Attachment (Using SNA On an X.25 PSDN)	6-7
Ensuring Changes	5-8	Binary Synchronous Communications Attachments (Point to Point)	6-8
Data Space Index Maintenance	5-9	Binary Synchronous Communications Attachments (Multipoint Tributary)	6-9
Materialization of Data Base Object Attributes and Statistics	5-9	Multi-leaving Telecommunications Access Method Support for MRJE	6-10
Modification of Data Base Object Attributes	5-9	System to System Attachment (SNA)	6-11
Recovery Considerations	5-9	System to System Attachment (for DHCF)	6-12
Performance Considerations	5-10	System to System Attachment (SNA)	6-13
Data Base Maintenance Functions	5-12	Configurations and States of Source/Sink Objects	6-14
Data Spaces	5-12	Configurations	6-14
Data Space Organization	5-14	Forward and Backward System Pointers	6-15
Data Space Indexes	5-14	Configurations Defined	6-15
Types of Addressing for Data Space Indexes	5-15	Configuration Information	6-17
Data Space Index Keys	5-15	Machine Configuration Record	6-17
Index Addressability to Subsets of Data Space Entries	5-17	Materialize Machine Configuration Record	6-17
Space Entries	5-17	Object Modification Limitations	6-17
Example of Data Space Index Ordering	5-17	Switched Network Considerations	6-17
JOURNAL MANAGEMENT	5-21	Switched Forward and Backward Pointers	6-18
Journal Objects	5-21	Network Description Candidate Lists	6-18
Journal Port	5-21	Controller Description Eligibility List	6-18
Journal Space	5-22	Object Contents	6-18
Specifying Objects to Be Journalled	5-22	Common Elements in LUD, CD, ND	6-20
Journal Entries	5-23	Specific Elements in LUD, CD, and ND	6-20
Applying Journalled Changes	5-24	Object States	6-20
Journal Status During IMPL	5-24	Source/Sink Instructions	6-26
Load/Dump	5-25	Instruction Usage	6-26
Commit Management	5-25	Exclusive Locks on Source/Sink Objects	6-26
Commit Object	5-25	Events	6-27
Commit Block	5-25	Create/Destroy Instructions—Hierarchy Rules	6-27
Commit Description	5-26	Configuration Hierarchy Rules	6-28
Commit Operation	5-26	Materialize Instructions	6-28
Decommit Operation	5-27	Modify Instructions	6-30
INDEX MANAGEMENT	5-29	LUD Session State Changes	6-33
Uses for Independent Indexes	5-29	Request I/O Instruction	6-37
Searching for Index Entries	5-29	Source/Sink Exceptions and Events	6-43
Inserting Index Entries	5-30	Communications Error Recovery	6-43
Performance Considerations	5-30	Systems Network Architecture Concepts	
QUEUE MANAGEMENT	5-31	for System/38	6-43
Queues	5-32	Application Layer (PGM)	6-43
Queue Instructions	5-32	Function Management Layer (PGM)	6-44
Queuing Functions	5-33	Transmission Management Layer (MI)	6-44
Moving Messages	5-34	SNA Transmission Management Layer	6-44
Materialize Queue Messages	5-34	Advanced Program-to-Program Communications	6-45
SPACE MANAGEMENT	5-35	Display Station Pass Through	6-46
Spaces	5-35	SNA Supervisory Services Support (MI)	6-48
Space Functions	5-35	Machine Services Control Point	6-48
Space Creation	5-36	MSCP Role (Primary Station)	6-49
Space Attribute Materialization	5-36	MSCP Role (Secondary Station)	6-50
Space Attribute Modification	5-36	MSCP Role (Peer Station)	6-51
Space Destruction	5-36	Binary Synchronous Communications Concepts	
Space Data	5-36	for System/38	6-52
Space Data Views	5-36	Application Layer (PGM)	6-52
Space Addressing	5-37	Function Management Layer (PGM)	6-52
Dump Space Management	5-37	I/O Management Layer (MI)	6-52
Dump Space Functions	5-37	X.25 Communications Concepts For System/38	6-53
Space Data Modification	5-39	Load/Dump Considerations	6-54
CHAPTER 6. SOURCE/SINK FUNCTIONS	6-1	LD Commands	6-55
Source/Sink Objects	6-1	Session Types	6-56
Object Types	6-2	Sequence of Operation	6-56
Local Device	6-3	REQIO (Request I/O) Instruction	6-56
Local Subsystem Devices	6-4	RD (Request Descriptor)	6-57
Remotely Attached Devices	6-5		
System to System Attachment (SNA)	6-6		

Modify LUD Sessions for LUD	6-61
LD Error Processing	6-61
Processing an MODLUD (Reset) Instruction	6-62
Feedback Record	6-62
Load/Dump Authority	6-62
Data Base and Load/Dump Networks	6-62
Load/Dump Performance	6-64
Load/Dump Journal Entries	6-64
Dumping and Loading Journal Spaces	6-65
Source/Sink Object Recovery	6-66
IPL Cleanup	6-66
Damaged Objects	6-66
Partial Damage	6-67
Partial Damage Recovery	6-67
Source/Sink Examples	6-68
Shared Usage of Source/Sink Objects	6-68
Configuration Changes	6-69
Activation of Switched Networks	6-78
Session State Changes	6-81
Request I/O Operations—Error Recovery Examples	6-82
CHAPTER 7. MACHINE SUPPORT FUNCTIONS	7-1
SYSTEM/38 SUPPORT FUNCTIONS	7-1
Machine Attributes	7-1
Machine-to-Programming Transition	7-1
Terminate Machine Processing Function	7-1
Machine Check Function	7-2
Machine Checks	7-2
Diagnostic and Service Functions	7-3
MACHINE OBSERVATION FUNCTIONS	7-4
Observation Functions	7-4
Inherent Machine Observation Functions	7-4
Trace Functions	7-4
Inherent Machine Observation Instructions	7-4
Tracing	7-5
Materialize Instructions	7-5
RECOVERY FUNCTIONS	7-6
Recovery Capabilities	7-6
Data Base Recovery Capabilities	7-6
System Recovery Capabilities	7-7
GLOSSARY	G-1
INDEX	X-1

The purpose of this publication is to help the reader gain an understanding of the functions provided by the System/38 machine interface. The level of information contained in this publication is above that of the individual instruction operational characteristics; therefore, the reader is expected to be familiar with programming in both machine and high-level languages. Individual instructions are included but only for the purpose of explaining their major function. (The details for each instruction are included in the *System/38 Functional Reference Manual*.)

This publication contains the following major parts:

- Chapter 1, *Introduction*, contains general information about the System/38 machine interface.
- Chapter 2, *Object Control Functions*, introduces System/38 as an object oriented system. An overview of the program objects and system objects, how to address these objects, and the authorization required to use them are also contained in this chapter.
- Chapter 3, *Program Functions*, defines the components of a program. Program functions contains information about the two logically distinct operations that occur during the execution of a program. This chapter also contains information about the computational and the branching instructions.
- Chapter 4, *Supervisor and Control Functions*, contains information about the event management functions and the instructions used to monitor the occurrence of a set of events and take action based on the occurrence of some or all of that set of events. This chapter also provides the user with information about: managing certain machine conditions called exceptions, the contents of a process and how to manage a process, and the system resources and the available facilities to manage these resources.
- Chapter 5, *Data Functions*, contains information about the operations required to address and use data stored in the data base. This chapter also describes data spaces, data space indexes, keys, and access techniques.
- Chapter 6, *Device Support Functions*, provides information about the instructions that are used to control devices. These instructions manipulate and control the I/O devices, manage the attachment network facilities, and define the configuration details of the system.
- Chapter 7, *Machine Support Functions*, provides information about the instructions that assist in problem determination and system observation.

This publication also contains a glossary of System/38 instruction terminology.

Note: This publication follows the convention that *he* means *he or she*.

SUMMARY OF CHANGES

Miscellaneous changes have been made throughout this manual. Information has been added to this manual to support the following:

- Dump space object
- Scan instructions
- Cipher instructions
- Terminate instruction
- Dump space management
- X.25 communications
- Distributed host command facility (DHCF)

WHAT YOU SHOULD KNOW

You should know the information contained in the *IBM System/38 Introduction*, GC21-7728, because it provides a summary of what the IBM System/38 is and how it can be used to meet the data processing needs of an organization.

IF YOU NEED MORE INFORMATION

Communications

- *IBM System/38 Data Communications Programmer's Guide*, SC21-7825
 - Describes how to configure communications lines, control units, and devices for use in binary architecture (SNA) communications.
 - Identifies the configuration commands used to define communications lines, control units, and devices for SNA or BSC communications.
 - Describes the use of data description specifications (DDS) for communications. Also, describes the Create Communications File (CRTCMNF) and Create BSC File (CRTBSCF) commands.
 - Identifies communications programming considerations and contains examples of RPG III and COBOL programs for communications with BSC and SNA devices and systems.
 - Describes error handling for communications.
 - Identifies the device-dependent considerations for supported SNA hosts, and BSC devices and systems.

Device Operation

- *IBM System/38 Operator's Guide*, SC21-7735
 - Describes operator/service panel controls and indicators
 - Describes system console screen and keyboard
 - Describes diskette magazine drive controls
 - Describes how to handle diskettes
 - Describes how to load and unload diskettes and diskette magazines
 - Describes how to remove diskettes from the drive station
 - Describes error recovery for printers

Problem Handling

- *IBM System/38 Diagnostic Aids*, SY21-0584
 - Describes problem symptoms and causes
 - Describes initial problem determination

Machine Interface Programming Information

- *IBM System/38 Functional Reference Manual—Volume 1*, GA21-9331
 - Describes functions that are performed by each machine interface instruction.
 - Identifies information needed to code each machine interface instruction.
 - Describes events and exceptions that are signaled by each machine interface instruction.
 - Describes attributes and specifications of the ODT (object definition table), ODV (ODT directory vector), and OES (ODT entry string).
- *IBM System/38 Functional Reference Manual—Volume 2*, GA21-9800
 - Describes input/output devices
 - Describes communications line connections
 - Describes load/dump functions
 - Describes machine initialization

The high-level machine interface of System/38 is the primary characteristic that identifies this system as a major advance in computer system architecture. In addition, many of the basic supervisory and resource management functions previously found in computer operating systems are included in the System/38 instruction set.

Note: In this book the terms machine interface and instruction set are interchangeable.

The object-oriented architecture of the interface is fundamental to the overall design of the functions provided by System/38. An object is a named entity that is described by its set of attributes. (The attributes define a set of functions or operations that can be performed on the object.)

The high-level operations performed by System/38 instructions provide the desired logical functions without dependence on their machine implementation. The power of these instructions is illustrated by data base operations that retrieve, update, and logically sort data records.

The access path to objects is machine controlled. This feature permits effective authority enforcement and automatic serialization of concurrent operations on the same object. Pointers, which are used to address objects, cannot be counterfeited. This feature prevents unauthorized addressability to objects or to virtual storage. These two features provide greater data integrity and security than available on previous systems.

System/38 executes each user's programs as an independent process. The machine resources (processor, storage, devices) that are shared by processes are also managed by the machine. Interprocess communication is accomplished through queues and event signals. Locks can be applied to objects to control and serialize concurrent access to those objects being shared by several processes.

Programs are translated into microcode to achieve greater efficiency. Program variable attributes cause the machine to automatically perform data-type conversions and allocate program work storage for these variables.

Input/output operations offer greater device independence through the use of the source/sink function and SNA (systems network architecture). The intricacies of the channel, communication networks, and asynchronous device operations are handled by the machine.

System/38 incorporates all of these features, and more, into the machine hardware and microcode. This high level of function is standard on every machine model regardless of storage size, processor type, or device configuration.

OBJECT CONCEPTS

Typical machine instruction sets for traditional systems provide bit and byte string manipulation capabilities. The System/38 instruction set provides similar functions and also provides machine instructions that operate on complex data structures to accomplish high-level functions.

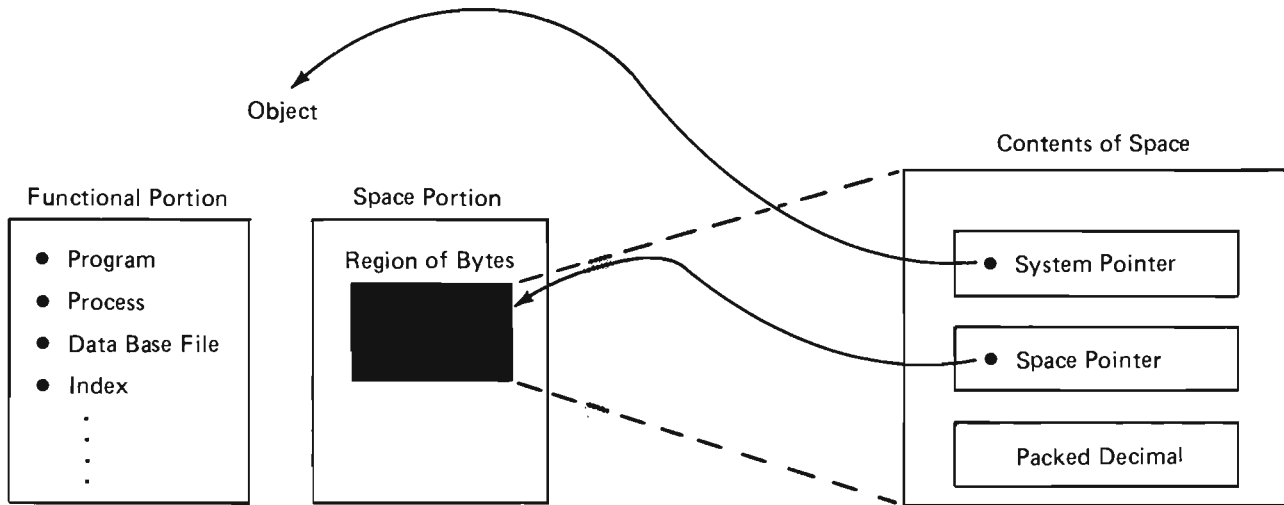
In System/38 some of the data structures, such as programs and data files, are similar to the programs and data files in conventional systems. Other data structures are unique to System/38. The data structures that are presented in the instruction interface are collectively categorized as objects.

Most objects must be created through use of a Create instruction. Other objects are implicitly created by the machine. Through a template, the user provides a set of attributes and values that apply to the new object. The new object also has operational characteristics that define the set of functions that can be accomplished through it.

An object consists of a functional portion and an associated space. The functional part of an object is used to implement a particular construct. For example, the functional part of a program object is created by the translation of System/38 instructions into microcode. The program is said to be encapsulated because there is no direct access to the storage that is used to support the program. Instead, the object is manipulated at a high level through the instruction set. In this way, encapsulation ensures the functional integrity of all objects.

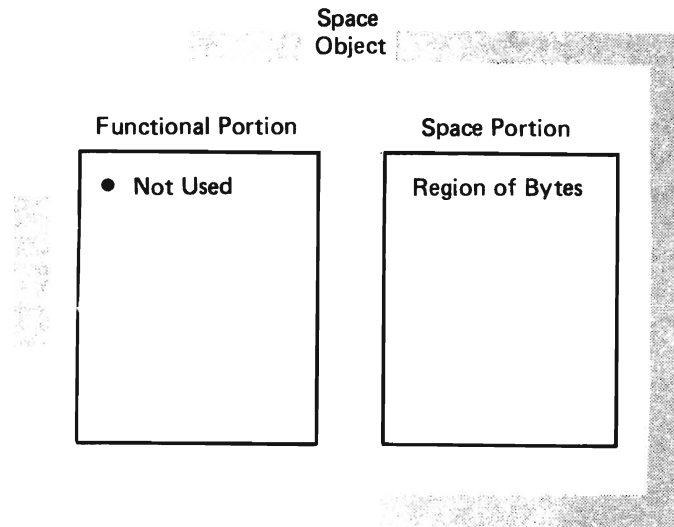
The associated space portion of an object is a region of bytes that can be directly manipulated by the user. The space is associated with the functional part of the object and provides a convenient way of storing additional user-defined data that is pertinent to the usage of that object.

Addressability to an object is obtained when the object is created or when the symbolic name of an object is resolved to form a pointer. (A pointer is data that is used for addressing either objects or bytes in spaces.) A system pointer, for example, enables a user to address an object for the purpose of destroying, materializing, or modifying an object through the instructions associated with that object type. A space pointer provides addressability to bytes within a space object or associated space.



Pointers are controlled through pointer manipulation instructions. (Pointer validity is maintained through a hardware protection mechanism.) A pointer is invalidated when a computational instruction is used to modify that pointer.

One type of object, called a space object, has no functional part. Its associated space is used to provide storage for control blocks, buffers, pointers, and other data.



Users need not be concerned with the addressing structures of main storage or auxiliary storage, or aware of multiple levels of storage, because the storage used for all objects is allocated and managed by the machine. That is, except for a possible degradation of performance, it makes no difference in the System/38 instruction set where an object, or portions of it, resides. Thus, the total address space of System/38 consists of objects that are uniformly addressable by pointers.

Similar constructs shield the user from dependencies upon channel and I/O device addresses and low-level communication protocols.

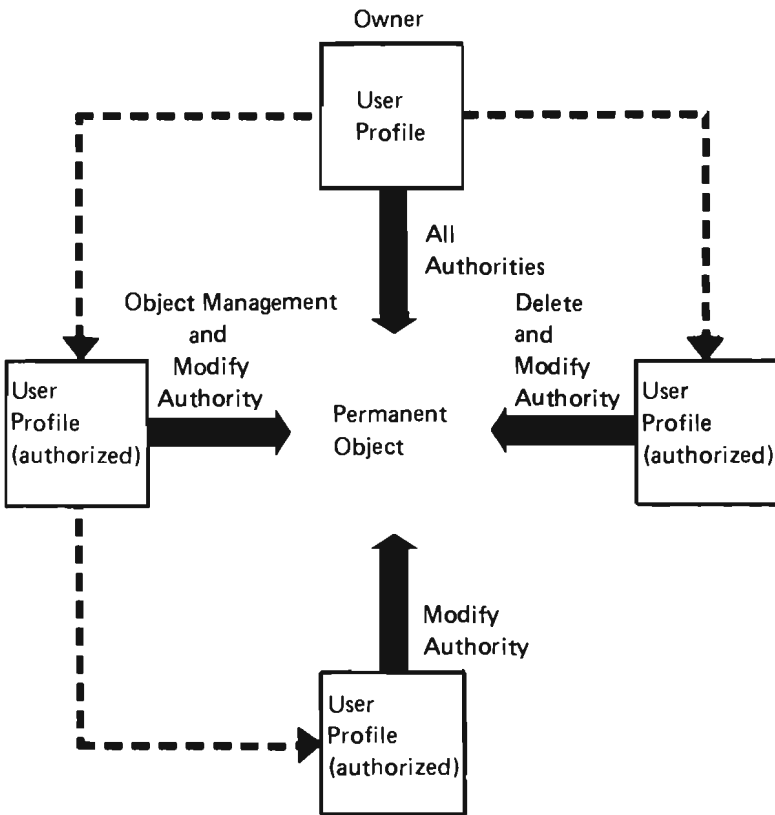
INTEGRITY AND AUTHORIZATION

Improved system integrity and authorization mechanisms is one advantage of the object-oriented approach. All user information is stored in objects. Access to that information is through System/38 instructions that ensure the integrity of the objects. An attempt to misuse an object is, therefore, detected and causes the execution of the instruction to be terminated and an exception condition to be signaled.

Authorization capabilities are likewise enhanced by the object-oriented interface. Each user of the machine is identified by a user profile, which is itself an object. Permanent objects in the system are owned by a user profile, and the owner or an appropriately authorized user profile may delegate to other user profiles various types of authority to operate on the objects.

Even though System/38 provides improved system integrity and authorization mechanisms, the user is responsible for overall controls and security. For these mechanisms to be effective, proper user implementation should be accompanied by other control practices, such as physical security and division of duties.

When implementing the security functions, special attention should be given to authorization of system-wide functions such as save/restore, creation of programs, and debug functions. Physical security considerations include the system console, save/restore diskettes, security officer password, and controls over machine-to-machine movements of data and programs.



Legend:

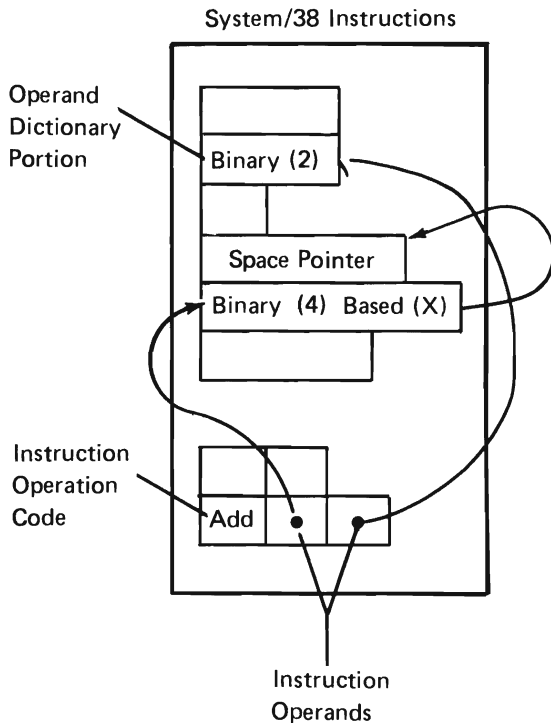
-----> = Delegates authority

—————> = Authority to object

A process (unit of multiprogramming) executes under control of a specific user profile (in the name of a user), and functions executed under the process verify that the referenced objects are properly authorized to that user.

System/38 instructions provide two modes of addressing. First, pointers allow varying addressability to objects and to bytes within space objects. Second, dictionary addressing deals with program references to values within a space object.

The operands in a program instruction are defined in a dictionary portion of the program that is separate from the instructions themselves. Instruction operands are index references to these dictionary entries that define the operand characteristics such as data type and length.



Binary, zoned decimal, packed decimal, character, and pointer data types are examples of operand characteristics that can be defined. The dictionary entries do not contain the operand values. They do, however, control the general type of location characteristics, (for example, relative to the area addressed by a pointer or relative to the storage area allocated for program variables within the executing process).

Additional capability, relative to low-level instruction interfaces, is provided by having instructions refer to dictionary entries that describe operand characteristics. For example, the following high-level capabilities are provided:

- Computational instructions are generic with respect to data type and length. For example, there is only one Add Numeric instruction in the System/38 instruction set; the Add Numeric instruction operates on whatever data is defined in the ODT (object definition table). This enables the use of source and receiver operands of varying type, length, and decimal positioning with all conversions and scaling being performed by the machine.
- Arrays can be defined in System/38 programs. Instruction operands support array indexing to locate specific elements of the array.
- Because applications often allow operations on multiple formats of data, some System/38 instructions (for example, the copy instructions) support attribute binding during execution time.

In addition to these types of high-level data operations, the System/38 instruction set provides and, in some cases requires, functions intended to directly support programming constructs more directly than in traditional machines. For example, programs are invoked through Call and Return machine instructions. Machine-controlled argument and parameter functions provide communications from one program to another. Allocation and initialization of storage for program variables within a process is performed by the machine.

Multiprogramming is supported through the concept of processes. A process is similar to a task in other systems and is the basis for managing work in the machine. The user controls the number of processes currently initiated, the priority of each process, and the relationship of one process to another with respect to processor usage and storage usage. The machine then allocates the processor and storage resources based on these parameters as well as on the current status of the process.

This level of multiprogramming support provides the following advantages:

- A single resource management mechanism is applied to all processing across all system activities. This reduces overhead and provides better management of resources in a complex and dynamic environment.
- Efficient resource management mechanisms can be used by taking advantage of hardware characteristics without requiring hardware-related dependencies in the programming.

Similarly, System/38 instructions provide the basic functional building blocks for a high-function integrated data base. Data base objects are provided with a complete set of functions that support different access mechanisms, file sharing, record format definition and mapping, efficient record retrieval, update, add, and delete. This support allows, for example, a data base file structure, which maps a single logical file into records with multiple formats and content, to be defined. In addition, a single physical data base file can have multiple indexes (access paths) defined over it, all of which are concurrently updated when the file is changed. Users of the file can view the data in a form suitable to their application needs.

Data base support provides the same types of advantages as are provided by multiprogramming support: efficient management of resources across a multi-user environment without requiring programming dependency upon hardware and machine implementation details.

Chapter 2. Object Control Functions

Object Management

An object is a named entity whose attributes and functional location are either described by a data view (the definition of a program object) or created by a System/38 Create instruction (if a system object).

The following discussion defines the two classes of objects (system and program) used in System/38 instructions.

SYSTEM OBJECTS

System objects are explicitly created by the user in order to perform some operation defined by the machine. System objects can be known and addressed throughout the entire system. They generally have no relationship to one another (as viewed by the machine), and their existence is generally independent of one another.

The following objects are system objects:

- Access group
- Commit block
- Context
- Controller description
- Cursor
- Data space
- Data space index
- Dump space
- Index
- Journal port
- Journal space
- Logical unit description
- Network description
- Process control space
- Program
- Queue
- Space
- User profile

System Object Descriptions

The following is a brief description of each system object.

Access Group

Access groups are system objects that enable a user to specify (as a group) those system objects that are used together. Then, when required by an operation, the entire contents of the access group can be moved from one type of storage (auxiliary or main) to another, thereby optimizing the movement of objects. The movement of the access group can occur because of an explicit request by the user or an implicit request by the machine. A reference to an individual object contained in the access group causes only that object (or part of it) to be moved.

To be a member of an access group, a system object must be created into that access group.

For more information concerning access groups, refer to *Resource Management* in Chapter 4.

Commit Block

A commit block is a permanent system object that holds information concerning the changes made to objects under commitment control. The commit block contains a list of the objects under commitment control, a list of data space record lock identifiers, a list of objects that have undergone changes in the current commit cycle, and the commit ID.

Context

A context is an object that contains addressability by name, type, and subtype to other system objects; that is, a context relates the symbolic identification of an object to an internal machine representation of the location of the object.

The machine context is implicitly created and maintained by the machine. It contains exclusive addressability to all user profiles, permanent contexts, logical unit descriptions, controller descriptions, and network descriptions.

Other contexts can be created to user specifications by a Create Context instruction. A user-created context can address any system object except those system objects whose addressability is restricted to the machine context (see preceding paragraph).

Each object that is addressed by a context must have (within the context) a unique symbolic identification. This unique symbolic identification includes object type, object subtype, and object name.

System objects, other than those restricted to the machine context, need not be addressed by any context. If such addressability exists, it is limited to only one context.

The user can obtain addressability to a system object by using the name resolution function. This function causes addressability to a system object to be returned in a system pointer.

For more information concerning contexts, refer to *Context Management* later in this chapter.

Controller Description

A controller description is an object that represents either an I/O controller for a cluster of I/O devices or a station that attaches groups of communications devices over the same data link.

Controller descriptions are created for every communications station or device controller that can be attached to System/38. The object contains a description of the controller, a pointer to the logical unit descriptions associated with it, and the current status of the controller.

For more information concerning controller description objects, refer to *Source/Sink Management* in Chapter 6.

Cursor

A cursor is a system object used to provide access to the entries residing within a data space. A cursor is the user's only interface to these entries.

A cursor can directly locate any entry in a data space; a cursor can also indirectly locate any entry in a data space through a data space index. A cursor, then, serves as the interface for retrieving data from and storing data into the data space.

For more information concerning cursors, refer to *Data Base Management* in Chapter 5.

Data Space

A data space is a system object that serves as the basic unit of storage for a user's data. A data space consists of a collection of entries, each of which may contain a given number of similarly formatted data fields.

A data space entry is the lowest level that can be addressed in accessing the contents of the data space. Each field in an entry is described by scalar type (binary, zoned decimal, packed decimal, and character) and length. When entries are retrieved or modified, the scalar attributes and length are used to transform the data to or from the format of the data presented in the user's program.

For more information concerning data spaces, refer to *Data Base Management* in Chapter 5.

Data Space Index

A data space index is a system object that is used to provide a logical ordering of the entries in a data space. Through use of a data space index, the entries in a data space can be accessed independently of the physical organization of the data. A data space index can be created for one or more data spaces for which a relationship can be expressed through a collection of key fields. The key field consists of one or more data fields that are selected from the data space entries to specify an ordered sequence.

For more information concerning a data space index, refer to *Data Base Management* in Chapter 5.

Dump Space

A dump space is a system object that serves as a storage area for a dump of other system objects. It provides an online storage alternative to the commonly used offline storage media for dumps of system objects.

A dump space is used to distribute the dumped objects it contains to other systems, but does not require offline media (such as tape or diskette). A dump space provides temporary backup of the dumped objects it contains. Because a dump space resides in the online storage of the machine, it has the same exposure to damage or loss due to failures of online storage that other system objects have. Therefore, a dump of system objects performed to provide a backup for recovery purposes can be performed from the dump space or directed to offline media.

A dump of system objects can be set into a dump space through a source/sink dump operation. Dump data can then be retrieved and inserted into another dump space on the originating system or on another system. The system objects in a dump contained within a dump space can be loaded back into a usable state on the machine through a source/sink load operation.

For more information concerning dump space objects, refer to *Dump Space Management* in Chapter 5.

Index

An index is a system object that can be used for storage and retrieval of data based on some key value. Scalar values and pointers can be inserted into the index with a portion of the value interpreted as the key field. The key value can subsequently be used to locate and retrieve one or more entries from the index.

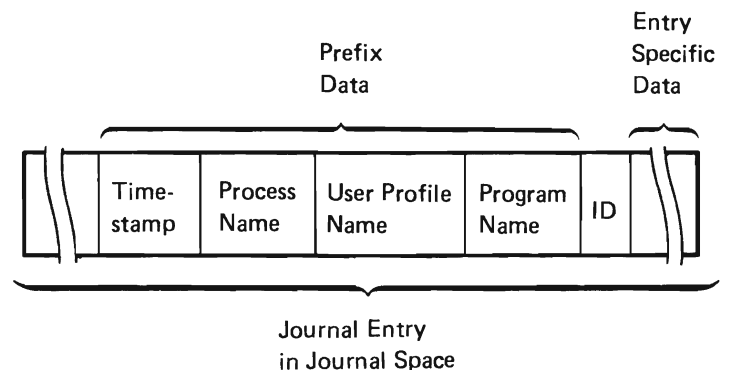
The index can be used for table look-up functions such as symbol tables, cross-reference lists, or dictionaries.

For more information concerning index objects, refer to *Index Management* in Chapter 5.

Journal Port

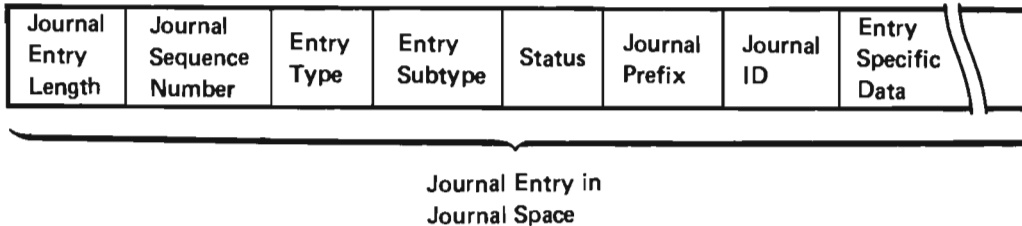
A journal port is a system object used to link objects to journal spaces. Only those objects having their changes journaled are linked to a journal space.

A journal port contains the definition of the prefix data associated with each change entry in the journal space.



Journal Space

A journal space is a system object used to contain the changes to those objects specified as journaled objects. When a journal space is attached to a journal port, all changes that occur to the objects attached to that journal port are sequentially entered in the journal space. These change entries are of variable length.



The user may insert entries in the journal space by using the Journal Data instruction.

The user may read the changes through the Retrieve Journaled Entries instruction.

Logical Unit Description

An LUD (logical unit description) is an object that represents a physical I/O device or an end-use mechanism such as another program in the device or system represented by this LUD.

A logical unit description must be created not only for every I/O device attached to the machine, but for every end-use mechanism that will communicate with this machine. This system object contains unique information to identify the device and determine its current status. The object is also used to control the I/O operations for that device.

For more information concerning logical unit descriptions, refer to *Source/Sink Management* in Chapter 6.

Network Description

A network description is an object that represents a communications network port on the system. A network description object must be created for each communications port on a machine.

For more information concerning network description objects, refer to *Source/Sink Management* in Chapter 6.

Process Control Space

A process control space is a system object that is used by the machine to control process execution.

The user must create a process control space before the initiation of a process. The process control space is then associated with that process for the entire period that the process exists. The machine uses the process control space for machine work areas and storage related to process execution.

For more information concerning process control space, refer to *Process Management* in Chapter 4.

Program

A program is a system object that forms the basic executable unit of the machine. The execution of a program causes a series of functions to be performed against a set of objects. A program is the encapsulated and executable form of a program template and logically contains both the function definition and object definitions from the program template.

The program can be executed once it is activated (storage is allocated and initialized for static data) within a process and invoked (storage is allocated and initialized for automatic data).

For more information concerning programs, refer to *Program Management* in Chapter 3.

Queue

A queue is an object that can be used for storage and retrieval of data, called messages. Messages can be stored (enqueued) in a queue for later retrieval (dequeued) by the same process or by a different process.

A process can test for a message on a queue and either wait or continue execution if the message is not available.

Messages can be inserted on and removed from a queue based on a key value, or they can be processed based on order of arrival, either FIFO (first-in-first-out) or LIFO (last-in-first-out).

For more information concerning queues, refer to *Queue Management* in Chapter 5.

Space

A space object contains only a space. It is used to store scalars and pointers and serves no other purpose.

A space is implemented with special hardware tag bits that identify the data in the space as a pointer. The machine instructions that establish, copy, and modify a pointer ensure that the pointer is valid by turning on, maintaining, and checking these tag bits. If a machine instruction overlays the pointer data in the space (for example, with a Copy Bytes instruction), the tag bits are turned off, thereby invalidating the pointer. This prevents any illegal use of a pointer and also ensures that a pointer is not counterfeit. (An attempt to use this data as a pointer results in an exception, and the instruction is not completed.) This scheme provides data integrity and security because pointers can contain authorization information as well as addresses.

For more information concerning space objects, refer to *Space Management* in Chapter 5.

User Profile

A user profile is the system object that identifies a valid user to the machine. A user profile also defines the user's rights of use to system objects, machine resources, privileged instructions, special machine functions, and certain machine attributes.

Each process executes under (or in the name of) a user profile (user). This user must have authorization to perform the various functions specified in the process. The user profile provides the mechanism for granting this authorization and for verifying that proper authorization exists.

For more information concerning user profiles, refer to *Authorization Management* later in this chapter.

System Object Characteristics

The following list is a summary of the characteristics of system objects:

- Each system object must be explicitly created. There are specific Create instructions (for example, Create Program and Create Context) for each type of system object.
- The Create instruction references a user-supplied template. The template provides a set of attributes and values to be assigned to the system object when it is created.
- System/38 instructions enable the user to determine the current attributes of the object.
- All system objects have a name. The name is part of a symbolic address that can be used to locate the object.
- System objects are created as either temporary or permanent objects.

Temporary system objects are implicitly destroyed when an IPL (initial program load) is performed unless they have already been destroyed by the user.

Permanent system objects must be explicitly destroyed by the user because they are not implicitly destroyed by the machine.

- All system objects (except temporary contexts) can be addressed by another system object called a context. (A context relates the name of an object with the address of that object.)

Certain types of system objects (for example permanent contexts, user profiles, controller descriptions, logical unit descriptions, and network descriptions) can only be addressed by the machine context.

All other system objects (except a temporary context) can be addressed from a temporary or permanent context or from no context at all. For these objects, the user may specify (as a creation attribute) a context that is to initially address the object. If a context is not specified, addressability is not placed in any context. The user may subsequently insert object addressability into a context, move object addressability from one context to another, or delete object addressability from a context. A system object cannot be addressed by more than one context.

- A system pointer provides addressability to a system object.

In order to reference a system object for any operation, a system pointer must be specified as an operand of the instruction or implied as a field in a template.

A system pointer can be created by using the name resolution functions provided through contexts (symbolic address) or using machine functions that return a system pointer to an object (for example, create, materialize, and retrieve functions).

- System objects can be explicitly destroyed by using the appropriate destroy instruction (for example, Destroy context), or temporary system objects can be implicitly destroyed by the machine.

- All system objects can have an associated space that is used to store pointers and data. The contents of the space are defined by the user and may be set or tested by any of the operations that operate on space data.

Spaces associated with system objects are located through a reference to the system object. Such spaces exist only as long as the containing system object exists.

The size of the associated space is defined when the system object is created. If the space is defined as variable in length, it may be extended or truncated at some later time.

- Temporary system objects can be placed in access groups. Access groups (which are also system objects) allow system objects that are used together to be transferred as a group from and to different types of storage. Access groups cannot, in turn, be contained in access groups.
- The right to use a system object is monitored by the machine. The authorization management functions verify specified users' rights of use to the object. An object can have an owner (identified by a system object called the user profile) who is initially granted all rights of use to the object; however, the object authorization rights of the owner may be retracted. Only appropriately authorized users can grant authorization rights to other users.

The machine prohibits the use of the object unless the proper authority has been granted.

- Object use can be synchronized through use of object locks. Locks are enforced based on the operation to be performed on the object. Locks are not required in order for an object to be referenced by an instruction. However, a process is not allowed to use an object if any other process holds a conflicting lock.

Common Attributes of System Objects

Each system object is created based on the attributes defined in a template. The format of the template is unique for each type of system object with each template containing a different set of attribute types. However, each template also contains a set of attributes that are common to all system objects. These common attributes are:

- System object identification
- Existence
- Space
- Context addressability
- Access group membership
- Performance class

The format of each template and the unique attributes for each object are described in the *System/38 Functional Reference Manual*. The common attributes, in general, are not discussed with each instruction but are described in the following text.

System Object Identification

System object identification provides a symbolic identification that is associated with the object. The object identification is the basis for the symbolic address of the object and can be used to locate the object through a context.

Object identification is contained in a 32-byte string. Object type is a 1-byte field implicitly assigned by the machine during object creation. Object subtype is a 1-byte field supplied by the creator of the object. The value in the subtype field can be used, based on user convention, to further qualify the object type; any byte value is allowed. Object name is a 30-byte entry assigned by the user during object creation. Any byte values are allowed in the name entry.

The object subtype and object name can be modified by the Rename instruction.

Existence

The existence attribute specifies whether the object is to be a temporary or permanent object. A temporary object, if not explicitly destroyed by the user, is implicitly destroyed when an IPL is performed. A permanent object exists in the machine until explicitly destroyed by the user.

Space

A space can be associated with the created object. The size of the space can be either fixed or variable. The initial allocation is specified in the size of space entry. The machine then allocates a space at least the size specified (the actual size is dependent on an algorithm defined by a specific implementation). A fixed size space of zero length causes no space to be allocated.

Each byte of the space is initialized to a value specified by the initial value of space entry. When the space is extended in length, this byte value is also used to initialize the new allocation.

Context Addressability

Addressability to a newly created object can optionally be inserted into a context as part of object creation. (Objects that are addressable by the machine context always have their addressability implicitly inserted into that context.)

Access Group Membership

Access groups contain only temporary objects and are used to store objects so that the objects that are used together can be moved between main storage and auxiliary storage as a group. Objects are placed into an access group during object creation.

Performance Class

The performance class attribute provides information that allows the machine to more effectively manage the object. The values allowed and their significance are attributes discussed in the *System/38 Functional Reference Manual*.

PROGRAM OBJECTS

A class of objects called program objects is defined in the ODT (object definition table). During execution of the Create Program instruction, the ODT, the instruction stream, and the other components of the program template are presented to the Create Program instruction. This instruction creates the program and makes the objects defined in the ODT part of that program. The ODT entry for a program object is, therefore, a declaration for the object rather than the actual object.

The entry in the object definition table that describes a program object is called a view. A view defines the type, attributes, functional location and, possibly, a permanent value or an initial value of the object.

An ODT contains a view for each program object to be used in a program. Each ODT entry contains the view for a single program object. System/38 instruction operands reference these views contained in the ODT. This means that the instruction is to perform a specified function by using one or more program objects referenced as operands.

Functionally, the created program contains two major parts, the function definition (from the instruction stream) and the object definitions (from the ODT).

The ODT definition of an object contains no storage allocation for the object. The ODT definition does specify whether storage for the object is to be allocated as a part of program execution and when that storage is to be allocated. The created program maintains these storage allocation definitions but does not contain any allocated objects. If storage is to be allocated, the allocation is done as part of the activation of a program (for static storage) or as part of the invocation of a program (for automatic storage).

These objects are considered to be program objects:

- Data object
 - Scalar data object
 - Pointer data object
- Constant data object
- Entry point
- Branch point
- Instruction definition list
- Operand list
- Exception description
- Space pointer machine object

The following describes each program object that can be defined by a view, and describes the attributes that can be specified with that object.

Data Objects

A data object is a program object that provides operational and possibly representational characteristics to byte strings in spaces for use as instruction operands. Data objects are defined in the ODT by data views. The data view establishes the attributes of the data object, its functional location, and its initial values. The smallest data object that can be mapped onto a byte string and addressed directly is a data element. A data object can be an aggregate of identical data elements to be operated on as a group (this is a data array). The two general classes of data element attributes are defined as scalar data attributes and pointer data attributes.

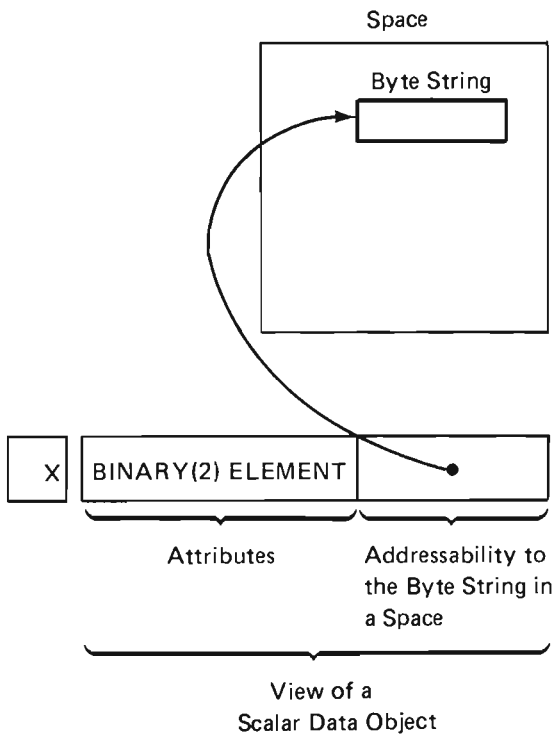
Scalar Data Objects

Scalar data objects provide support for operations on values in spaces. The scalar object provides the byte string it is mapped to through a scalar view with representation and operational characteristics.

For example, assume a scalar data object with the following definition:

X BINARY(2) ELEMENT

A reference to X assumes the storage addressed by X contains a value stored as a 2-byte binary number.



Only scalar data objects can have a bit or character string representation defined. By defining a numeric scalar object and a character scalar object onto the same byte string, the user can manipulate the character scalar object and get a predictable result on the numeric scalar object. Instructions that reference scalar data objects depend on proper representations and likewise produce results that have predictable representations.

Scalars can be numeric or character types. Numeric type scalars include binary, zoned decimal, and packed decimal.

Binary Elements: Binary elements have the following characteristics:

- Binary elements are stored as a series of 16 or 32 binary digits (0 or 1).
- Binary elements represent signed values with the sign encoded in the leftmost binary digit.
 - A leftmost binary digit of 0 represents a positive value
 - A leftmost binary digit of 1 represents a negative value
- Two binary lengths are defined:
 - Binary (2)–(Two bytes or 15 binary digits for the value and one sign digit.)
 - Binary (4)–(Four bytes or 31 binary digits for the value and one sign digit.)
- The range of values for binary numbers is:
 - Binary (2) -2^{15} to $2^{15} - 1$
 - Binary (4) -2^{31} to $2^{31} - 1$
- Negative binary values are stored in twos complement form.

Zoned Decimal Elements: Zoned decimal elements have the following characteristics:

- Zoned decimal values are stored as a series of zoned decimal digits.
- Zoned decimal digits are stored in 8-bit bytes and are encoded with a 4-bit zone in the leftmost four binary digits of the byte and a true form decimal digit encoded in the rightmost four binary digits of the byte.

- The sign of the field is encoded in the zone field of the rightmost byte of the value. For example, a positive decimal value of 5 678 is encoded as follows:

Byte	1	2	3	4
Hex Value	F5	F6	F7	F8
				↑
				Sign Position

- The following binary encodings are valid signs:
 - Positive–1111, 1100, 1110, 1010. The binary encoding 1111 is the preferred positive sign. (The preferred sign is the encoding that an operation, other than copy, uses for the result.) The Copy Numeric Value instruction also sets the preferred sign.
 - Negative–1101, 1011. The binary encoding 1101 is the preferred negative sign.
- Zoned decimal numbers can contain from 1 through 31 decimal digits with any number of these digits specified as fractional positions.
- The zone field in all bytes of a zoned decimal number, other than the rightmost byte, can have any value. During an arithmetic operation (including the Copy Numeric Value instruction), the zone field for all bytes of a result field, except the rightmost byte, is set to the binary value, 1111.

Packed Decimal Elements: Packed decimal elements have the following characteristics:

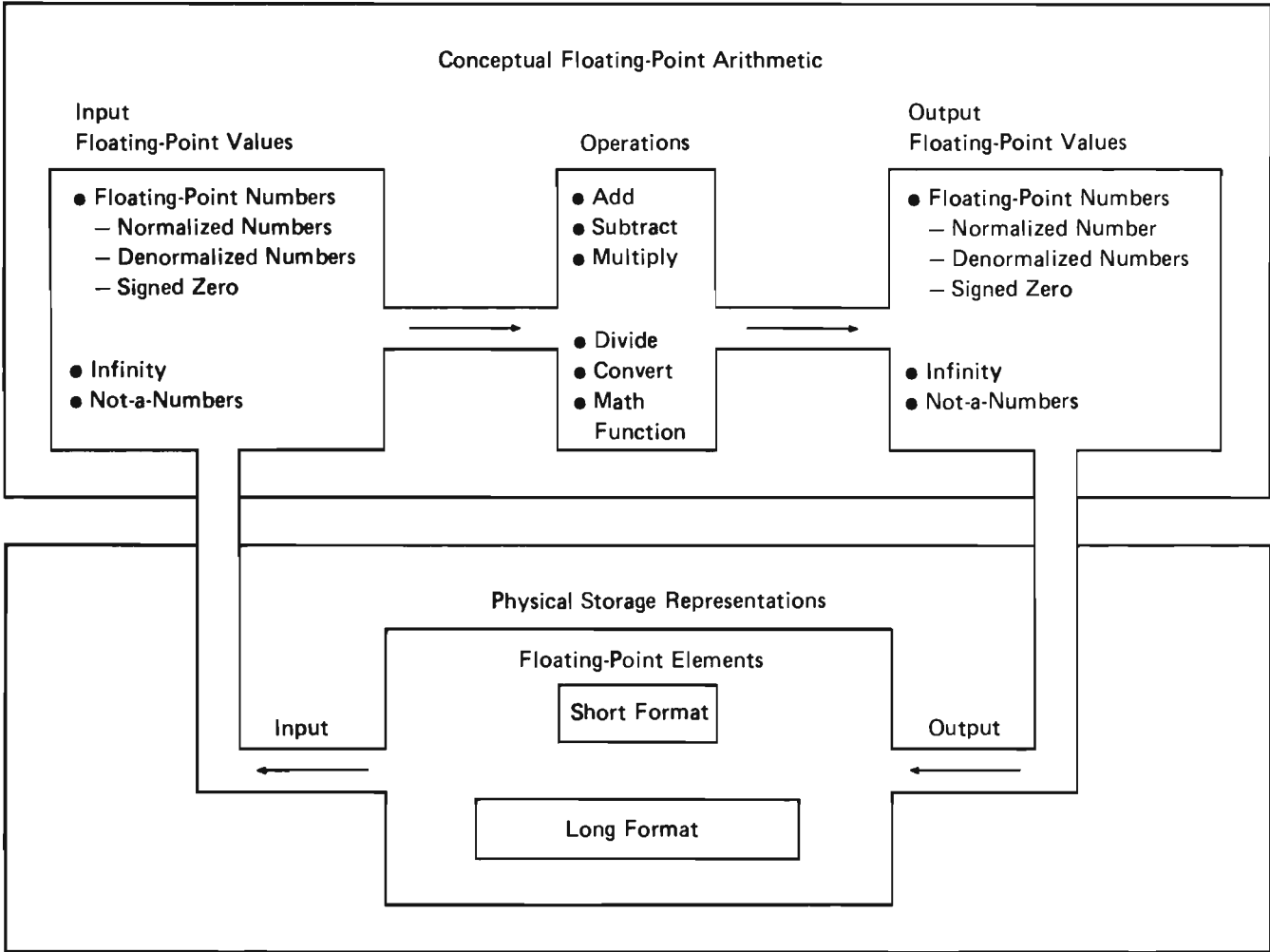
- Packed decimal numbers represent signed values and are stored as a series of decimal digits.
- Each decimal digit is encoded in true form as a 4-bit binary value. Consequently, two decimal digits can be accommodated in a single byte except for the byte that contains the sign. The algebraic sign of a packed decimal field is encoded as a 4-bit binary value and must appear in the rightmost four bits of the rightmost byte in the field. For example, a positive value of 1 234 in a packed decimal view can be contained in 3 bytes as follows:

Byte	1	2	3
Hex Value	01	23	4F
			↑
			Sign Position

- The following binary encodings are valid signs:
 - Positive–1111, 1010, 1100, 1110. The binary encoding 1111 is the preferred positive sign.
 - Negative–1101, 1011. The binary encoding 1101 is the preferred negative sign.
- Packed decimal numbers can contain from 1 through 31 decimal digits. A packed decimal value (31 digits) is stored in 16 bytes. Any number of these digits can be specified as fractional digits.

Floating-Point Elements: Floating-point elements contain the storage representations of binary floating-point values. As such, the elements exist in storage and have a physical storage format, but the binary floating-point values are conceptual because they are the values operated on or produced by mathematical calculations performed in floating-point. A significant amount of the function performed in floating-point calculations relates to the implicit transformations that occur between a conceptual floating-point value and its representation, which exists in a floating-point element. In this sense, it is important to understand the concept of floating-point values as contrasted with their physical representation in storage.

The following diagram illustrates the concept of binary floating-point arithmetic and the physical binary floating-point storage elements.



Binary Floating-Point Values

This section describes the characteristics of the binary floating-point values, which the floating-point elements represent. A binary floating-point value is one of the set of values supported in binary floating-point calculations. This set of values is composed of binary floating-point numbers, infinity, and not-a-numbers. Binary floating-point values have the following characteristics:

- A binary floating-point number is a conceptual numeric value that contains a signed significand and a signed exponent. Its numeric value is the signed product of its significand and 2 raised to the power of its exponent. In addition to the sign, the significand of a binary floating-point number is composed of binary digits, which contain one integer digit to the left of the binary point and one or more fraction digits to the right.

An arbitrary notation to express a binary floating-point number is

(snnnn NUM sb.bbbbbbbb)

where: snnnn specifies the signed exponent value as a signed decimal integer; NUM acts as a separator between the exponent and the significand; sb.bbbbbbbb specifies the signed significand as a signed binary number expressed as a variable-length sequence of binary digits (values of 0 or 1) with an embedded binary point between the first and second digits. For example,

(+nnn NUM -b.bbbbbbbb)

would specify a binary floating-point number of value,

(-b.bbbbbbbb * 2ⁿⁿⁿ)

where * denotes multiplication, and ⁿⁿⁿ denotes exponentiation.

Binary floating-point numbers are either normalized numbers, denormalized numbers, or signed 0.

A normalized floating-point number is a number whose significand integer digit has a value of 1. For example, a value of 2 is expressed as

(+1 NUM +1.0).

A denormalized floating-point number is a nonzero number whose significand integer digit has a value of 0 and whose signed exponent has a value of -126 or -1022. Denormalized numbers provide for a gradual underflow toward zero of numbers representable in a storage format. For example, a value of 2⁻¹²⁷ is expressed as

(-126 NUM +0.1).

Signed 0 is 0 with an associated sign. It can be thought of as having a significand of 0 and an exponent of 0. Although 0 has an associated sign, positive 0 is always considered equal to negative 0. For example, positive 0 is expressed as

(+0 NUM +0.0) or as (+0).

- Infinity is a conceptual value that has an associated sign and is mathematically greater in magnitude than any binary floating-point number. An arbitrary notation can be used to express infinity where: +infinity specifies positive infinity and -infinity specifies negative infinity.

- Not-a-Number (NaN) is a conceptual value, not interpreted as a mathematical value, that contains a mask state and a sequence of binary digits. The mask state can be either masked or unmasked. The binary digits have no specific meaning other than to give the NaN a unique value. An arbitrary notation to express not-a-number is

(m/u NaN bbbbbbb)

where: m/u specifies the mask state (m for masked, u for unmasked); NaN acts as a separator between the mask state and the binary digits; bbbbbbb specifies the variable-length sequence of binary digits associated with the NaN. For example,

(m NaN 1010101010101010101)

specifies a masked NaN with a binary digit sequence of 1010101010101010101.

Floating-point elements have the following characteristics:

- Floating-point elements have a fixed-length storage format, which can be either a short (4-byte) format or a long (8-byte) format.

The short format, as shown by the following illustration, is a 32-bit string in which bit 0 is the sign field (S), bits 1 through 8 are the 8-bit exponent field (E), and bits 9 through 31 are the 23-bit fraction field (F).

Short Format

Sign (S)	Exponent (E)	Fraction (F)
0	8	31

The long format, as shown by the following illustration, is a 64-bit string in which bit 0 is the sign field (S), bits 1 through 11 are the 11-bit exponent field (E), and bits 12 through 63 are the 52-bit fraction field (F).

Long Format

Sign (S)	Exponent (E)	Fraction (F)
0	11	63

- The sign field value indicates the sign of the significand of the numeric value represented or the sign of the infinity value represented. A value of 0 indicates a positive sign. A value of 1 indicates a negative sign. The sign field does not have a defined meaning for NaN representation.
- The exponent field value indicates which type of binary floating-point value is represented. The exponent field value is a binary integer value; it contains either a reserved value or a biased exponent. A reserved value is one of the set of maximum and minimum exponent field values of 0 and 255 for the short format or 0 and 2047 for the long format. These values identify representations of denormalized numbers, signed 0, infinity, and NaNs. A biased exponent is one of the set of values between the maximum and minimum exponent field values. The range of biased exponent values is 1 through 254 for the short format or 1 through 2046 for the long format. These values identify representations of normalized numbers.
- The fraction field contains a sequence of binary digits whose meaning is dependent upon the binary floating-point value represented.

Both the short and long formats provide for representation of numbers, NaNs, or infinity. The bit encoding of the representations vary with the binary floating-point value represented and have the following characteristics:

- A normalized number is represented when the exponent field contains a biased exponent. The biased exponent is the non-negative sum of the signed exponent of the represented number and a constant value (bias). The bias is 127 for the short format or 1023 for the long format. The signed exponent of the number represented is calculated by subtracting the bias from the biased exponent value. The significand of the number represented has an integer value of 1, which is implied, and a fraction value from the fraction field. For example, the normalized number (+1 NUM +1.0), which is the value 2, is represented in the short format by hex 40000000.
- A denormalized number is represented when the exponent field contains a reserved value of 0, and the fraction field contains a nonzero value. The significand of the number represented has an integer value of 0 which is implied, and a fraction value from the fraction field. The reserved value of 0 in the exponent field indicates that the value of the signed exponent of the number represented is -126 for the short format or -1022 for the long format. For example, the denormalized number (-126 NUM +0.1), which is the value 2^{*-127} , is represented in the short format by hex 00400000.
- Signed 0 is represented when the exponent field contains a reserved value of 0, and the fraction field contains all 0's. For example, the value +0 is represented in the short format by hex 00000000.
- Infinity is represented when the exponent field contains the format's maximum reserved value (255 for short format or 2047 for long format), and the fraction field contains all 0's. For example, the value -infinity, is represented in the short format by hex FF800000.

- Not-a-number (NaN) is represented when the exponent field contains the format's maximum reserved value (255 for short format or 2047 for long format), and the fraction field contains a nonzero value. The sign field value does not have a defined meaning.

The leftmost bit position of the fraction field, bit 9 in the short format or bit 12 in the long format, indicates the mask state. A value of 1 in this bit position indicates that the NaN is masked. A value of 0 in this bit position indicates that the NaN is unmasked. Unmasked NaNs in a floating-point operand force the detection of the invalid floating-point operand exception. Masked NaNs in a floating-point operation are moved into the result field, but do not force detection of the invalid floating-point operation exception.

The remaining bit positions of the fraction field contain the sequence of binary digits associated with the NaN. The values of these binary digits have no defined meaning.

For example, the not-a-number (m NaN 101010101010101010101) is represented in the short format by either hex 7FD55555 or hex FFD55555 because the value of the sign field has no meaning for the representation of a NaN.

The system default NaN returned for certain floating-point exceptions is a masked NaN with a sign field value of 0 and a fraction value of 1 in the leftmost bit position (the masked state), followed by all 0's. For example, hex 7FC00000 is the system default NaN for the short format, and hex 7FF8000000000000 is the system default NaN for the long format.

A NaN can represent the results of incorrect combinations of operands in floating-point operations. A potential usage of these NaN values is to set them into uninitialized floating-point fields. This allows the system to detect a reference to a field that has not been set with a value by the time it is accessed.

The following information provides a summary of the binary floating-point values that can be represented in floating-point elements. In the following formulas, S represents the sign field value; E, the exponent field value; and F, the fraction field value of a floating-point element as previously described. Additionally, the ** characters denote exponentiation, the x character denotes multiplication, and the ~ character denotes a logical not.

Short Format

The values that can be represented are:

- Normalized number
(For $0 < E < 255$, value = $(-1)^{**S} \times 2^{*(E-127)} \times 1.F$)
- Denormalized number
(For $E=0$ & $F \neq 0$, value = $(-1)^{**S} \times 2^{*(-126)} \times 0.F$)
- Signed zero
(For $E=0$ & $F=0$, value = $(-1)^{**S} \times 0$)
- Infinity
(For $E=255$ & $F=0$, value = $(-1)^{**S} \times \text{infinity}$)
- Not-a-number
 - (For $E=255$ & $F \neq 0$ (bit 9)=1, value = masked NaN)
 - (For $E=255$ & $F \neq 0$ & (bit 9)=0, value = unmasked NaN)

Long Format

The values that can be represented are:

- Normalized number
(For $0 < E < 2047$, value = $(-1)^{**S} \times 2^{**(E-1023)} \times 1.F$)
- Denormalized number
(For $E=0$ & $F \neq 0$, value = $(-1)^{**S} \times 2^{*(-1022)} \times 0.F$)
- Signed zero
(For $E=0$ & $F=0$, value = $(-1)^{**S} \times 0$)
- Infinity
(For $E=2047$ & $F=0$, value = $(-1)^{**S} \times \text{infinity}$)
- Not-a-number
 - (For $E=2047$ & $F \neq 0$ & (bit 12)=1, value = masked NaN)
 - (For $E=2047$ & $F \neq 0$ & (bit 12)=0, value = unmasked NaN)

The range covered by the magnitude (M) of floating-point numbers that can be represented is:

- In the short format:
 - Normalized numbers
$$2^{**-126} \leq M \leq (2-2^{**-23}) \times 2^{**127}$$
or
$$1.17549435 \times 10^{**-38} \leq M \leq 3.40282347 \times 10^{**38}$$
 - Denormalized numbers
$$2^{**-149} \leq M \leq (1-2^{**-23}) \times 2^{**-126}$$
or
$$1.40129846 \times 10^{**-45} \leq M \leq 1.17549422 \times 10^{**-38}$$
- In the long format:
 - Normalized numbers
$$2^{**-1022} \leq M \leq (2-2^{**-52}) \times 2^{**1023}$$
or
$$2.2250738585072013 \times 10^{**-308} \leq M \leq 1.7976931348623158 \times 10^{**308}$$
 - Denormalized numbers
$$2^{**-1074} \leq M \leq (1-2^{**-52}) \times 2^{**-1022}$$
or
$$4.9406564584124654 \times 10^{**-324} \leq M \leq 2.2250738585072009 \times 10^{**-308}$$

The following chart shows the formulas and hexadecimal representations for:

- Plus and minus normalized floating-point numbers
- Plus and minus denormalized floating-point numbers
- Plus and minus zero
- Plus and minus infinities
- Masked NaNs
- Unmasked NaNs

Short Format (4-bytes)	+infinity	Long Format (8-bytes)
Hex 7F800000		Hex 7FF0000000000000
No representation		No representation
Maximum $((2-2^{23}) \times 2^{127})$ Hex 7F7FFFFF		Maximum $((2-2^{52}) \times 2^{1023})$ Hex 7FEFFFFFFFFFFFFFFF
Normalized		Normalized
Minimum (2^{126}) Hex 00800000		Minimum (2^{1022}) Hex 0010000000000000
Maximum $((1-2^{23}) \times 2^{126})$ Hex 007FFFFF		Maximum $((1-2^{52}) \times 2^{1022})$ Hex 000FFFFFFFFFFFFFFF
Denormalized		Denormalized
Minimum (2^{149}) Hex 00000001		Minimum (2^{1074}) Hex 0000000000000001
No representation	+	No representation
(+0) Hex 00000000		(+0) Hex 0000000000000000
(-0) Hex 80000000	0	(-0) Hex 8000000000000000
No representation		No representation
No representation	-	No representation
Maximum $-(2^{149})$ Hex 80000001		Maximum $-(2^{1074})$ Hex 8000000000000001
Denormalized		Denormalized
Minimum $-((1-2^{23}) \times 2^{126})$ Hex 807FFFFF		Minimum $-((1-2^{52}) \times 2^{1022})$ Hex 800FFFFFFFFFFFFFFF
Maximum $-(2^{126})$ Hex 80800000		Maximum $-(2^{1022})$ Hex 8010000000000000
Normalized		Normalized
Minimum $-((2-2^{23}) \times 2^{127})$ Hex FF7FFFFF		Minimum $-((2-2^{52}) \times 2^{1023})$ Hex FFEFFFFFFFFFFFFFFF
No representation		No representation
Hex FF800000	-infinity	Hex FFF0000000000000

Hex 7FC00000 — Masked NaN minimum — Hex 7FF8000000000000
Hex 7FFFFFFF — Masked NaN maximum — Hex 7FFFFFFFFFFFFFFF
Hex 7F800010 — Unmasked NaN minimum — Hex 7FF0000000000001
Hex 7FBFFFFFFF — Unmasked NaN maximum — Hex 7FF7FFFFFFFFFFFFFFF
Note: Use of sign field bit value of 0 is arbitrary.

Pointer Data Objects

Pointer data objects provide addressability to both program and system objects.

Spaces are designed with special tag bits that identify the data in a space as a pointer. The machine instructions that establish, copy, and modify a pointer ensure that the pointer is valid by turning on, maintaining, and checking these tag bits. If a machine instruction overlays the pointer data in the space (for example, with a Copy Bytes Left-Adjusted instruction), the tag bits are turned off, thereby invalidating the pointer. This prevents any illegal use of a pointer and also ensures that a pointer is not counterfeit. (An attempt to use this data as a pointer results in an exception, and the instruction is not completed.) This scheme provides data integrity and security because certain types of pointers can contain authorization information as well as storage addresses.

Four pointer data types are defined: space pointer, data pointer, system pointer, and instruction pointer.

Space Pointer: An SPP (space pointer) provides addressability for based data objects. The entry in the ODT (object definition table) for the based data object can specify a particular space pointer as its base. The space pointer can then be changed in order to provide data object addressability to a specific location in a space.

A space pointer logically consists of a pointer to the first byte of the space and an offset value that addresses a specific byte within the space. (The space pointer can address any space and any byte within that space.) The Set Space Pointer instruction can refer to a space through a space pointer, a data object, or a data pointer to obtain this addressability. In addition, the offset value can be changed separately (set, stored, added, subtracted) by using special instructions that refer to the space pointer.

No data attributes are associated with the storage addressed through a space pointer. A reference to a space pointer has no representation and operational characteristics other than as a space pointer. However, for a based data object, the based data object provides the view, and the space pointer provides addressability.

Data Pointer: A data pointer contains scalar data element attributes and addressability to a location in a space. The attributes can define any of the scalar data and element attributes that can be specified for scalar data objects.

Data pointers provide a means of accessing external data objects for use as instruction operands.

A data pointer can be resolved to an external scalar data object or set to describe a local scalar data object.

System Pointer: A system pointer is used to address any system object.

A reference to a system pointer as an instruction operand refers to the system object addressed by the system pointer (except for the Create, and Resolve System Pointer instructions in which the system pointer acts as the receiver for the instruction).

Instruction Pointer: An instruction pointer can be used to define variable branch targets within the instruction stream for a single program.

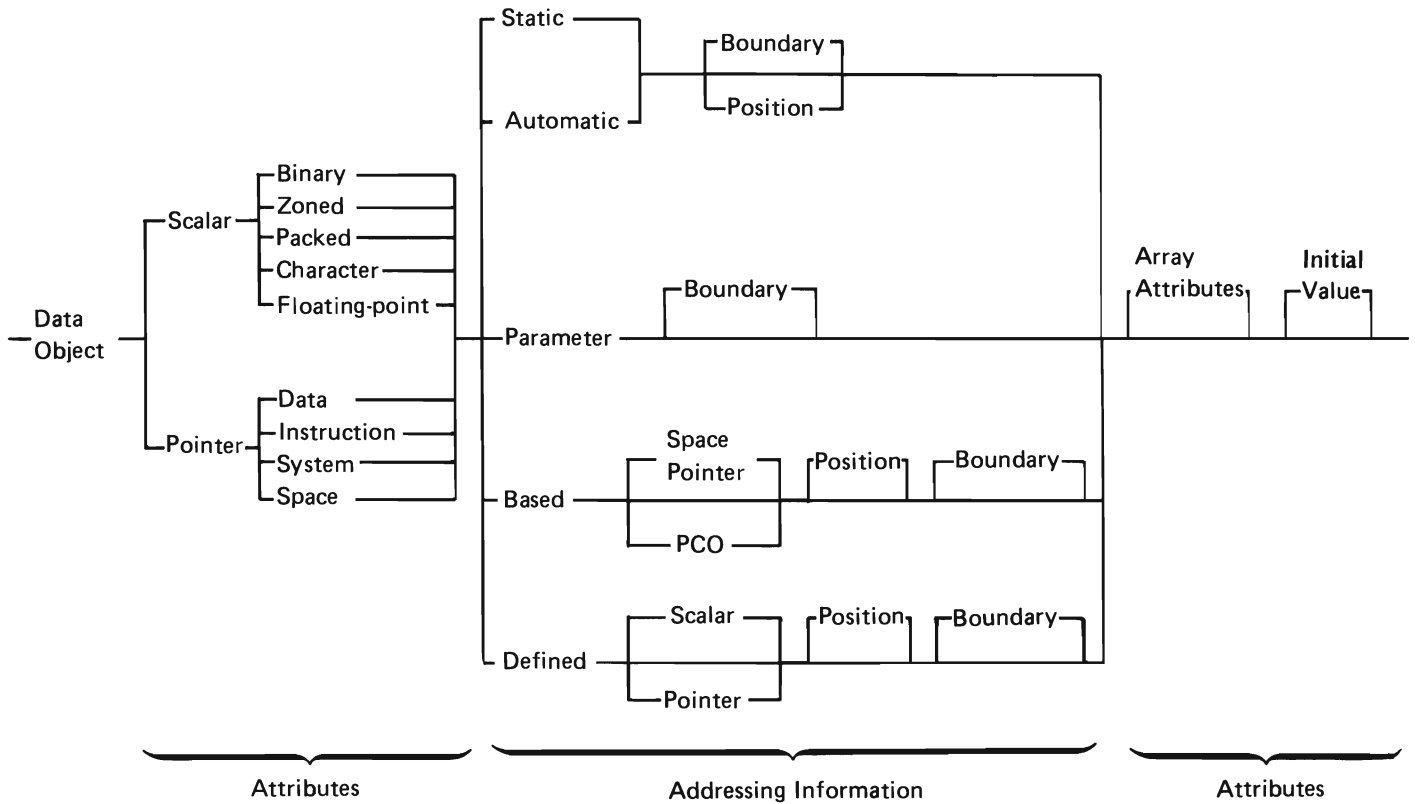
An instruction pointer can be set to address an instruction and, at some later time, execute a branch operation using the instruction pointer to define the target instruction. The instruction pointer can be set to an instruction number, a relative instruction number, or a branch point.

An instruction pointer can be used as a return target for an internal subinvocation. The Call Internal instruction sets an instruction pointer to address the next sequential instruction.

Common Attributes of Data Objects

A data object is defined through a data view that specifies a particular combination of data object attributes.

The following chart shows the valid attribute combinations for a data view of a data object.



The following major data object attributes are discussed as separate topics:

- Element and array attributes.
- The mapping attribute (direct on static or automatic, defined, parameter, and based).
- Initial values.
- External scalar data views.

All other attributes are discussed where appropriate.

Element and Array Attributes

Scalar data can be defined as elements (single elements) or arrays of elements (a grouping of data elements in which all elements have the same characteristics).

Data Element Views: A data view declared as an element describes the smallest data object that can be directly addressed.

An element has no directly addressable subelements. For example:

- X BINARY(4) ELEMENT

X represents a single binary number stored in 4 bytes. Anytime X is referred to, the entire 4-byte number is used.

- Y CHARACTER(12) ELEMENT

Y represents a single character string stored in 12 bytes. Any reference to Y refers to all 12 bytes.

- Z POINTER ELEMENT

Z represents a pointer object that is stored in 16 bytes (pointer objects are always contained in 16 bytes). Any reference to Z refers to all 16 bytes.

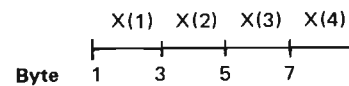
Arrays of Elements: An array view defines a data organization that is a collection of data elements, all of which have identical attributes. An element of an array view can be used as a scalar value or a pointer in an instruction. A compound array element operand selects a specific element of an array to be used in the instruction.

The number of elements contained in an array is specified in the array size entry in the data view; from 1 through 16 777 215 elements can be defined. Individual elements can be contiguous or noncontiguous. For contiguous arrays, the last byte of one element is immediately followed by the first byte of the next element. For noncontiguous arrays, individual elements are not immediately adjacent to one another. The number of bytes from the first position of one element to the first position of the next element is specified in the array element offset attribute and need not be the same number of bytes as is in an individual element. Noncontiguous arrays are allowed only for defined data views.

Assume the following declaration:

X BINARY(2) ARRAY(4)

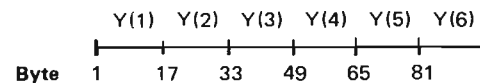
X would have the following storage mapping.



Assume the following declaration:

Y POINTER ARRAY(6)

Y would have the following storage mapping because a pointer is always contained in 16 bytes.



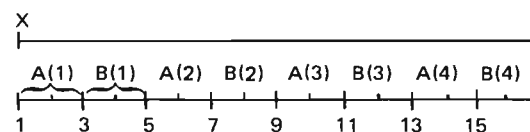
Assume the following declarations:

X CHARACTER (16)

A BINARY(2) ARRAY(4) ARRAY ELEMENT
OFFSET(4)
DEFINED(X) POSITION(1)

B BINARY(2) ARRAY(4) ARRAY ELEMENT
OFFSET(4)
DEFINED(X) POSITION(3)

The following mapping would result.



Data Object Mapping Attribute

Data views provide a mapping of a data object onto a byte string in a space. The mapping attributes (direct on static, direct on automatic, defined, based, and parameter) establish the functional location of the data object.

Direct Data Views (Static and Automatic): Direct data views provide a mapping relative to the static or automatic storage allocation. The functional location of the data object is dependent on the allocation of the static or automatic storage. A data view mapped direct on automatic, for example, causes the location of the data object to be determined relative to the allocation of the automatic storage in the PASA (process automatic storage area) for the process in which the program is invoked.

Defined Data Views: Defined data views provide a mapping relative to a previously defined data view. The functional location of the defined data view depends on the location of the base data view. The base data view can have a direct, based, defined, or parameter mapping attribute.

Based Data Views: Based data views provide a mapping relative to the addressability contained in a space pointer. The value of the space pointer, then, defines the functional location of the data object. The space pointer on which the data object is based can be optionally declared in the based data view. This declaration is for the duration of the program unless it is overridden for a single operand that specifies the explicit base compound operand. If a base is not declared, the based data view must be referenced in an explicit base compound operand where a space pointer is specified as a base for a single operand.

Parameter Data Views: Parameter data views provide a mapping that corresponds to an argument passed on a Call External, Call Internal, or Transfer Control instruction.

Data objects can be passed as arguments and received as parameters. If a data object is passed, the functional location of the parameter data object is the same as the argument data object.

The characteristics defined by the parameter data object are applied to the storage area defined by the corresponding argument. The functional location of the corresponding arguments is used, and no position value can be specified.

Initial Values

An initial value can be specified if the data object is located relative to a static or automatic storage allocation. This initial value overrides the original value assigned to the space and any previously declared data views that are positioned wholly or partially in the same space location.

The initial value can be specified as a single byte string of a length and format equivalent to that required to initialize the data view or, alternately, as one or more byte strings that are used repeatedly by the machine (based on user specification) to form the initial value for the data view.

The initial value for a space pointer can be specified as a data object to be addressed, or the initial value for an instruction pointer can be specified as an instruction number.

The initial symbolic value for a system pointer is specified as the name of the system object that the system pointer is to address. The name can be further qualified by specifying the name of the context to be searched for the system object. For more details on system pointer initial values, refer to *Address Resolution Functions* later in this chapter.

The initial value for a data pointer is specified as a symbolic value. This symbolic value specifies the name of the external scalar data object that the data pointer is to address. This name can be further qualified by specifying the name of the program to be searched for the data object. For more information about data pointer initial values, refer to *Data Object Address Resolution* later in this chapter.

External Attribute

A scalar data view mapped onto static storage can be declared as external. This specifies that the view can be known outside the declaring program.

The external scalar data view is assigned a name that becomes the symbolic address of the external data object. When this name is qualified by the name of the containing program, the fully qualified name of the external data object is formed. A data pointer in another program can gain addressability to the external data object by using the name resolution functions. This data pointer then adopts (assumes) the scalar data object attributes (functional location and current value) of the external scalar data object. A reference to the data pointer is functionally identical to a reference to the external scalar data object.

Constant Data Objects

A constant data object is a program object that defines a scalar data view that remains the same throughout the existence of a program.

The same scalar attributes that can be specified for a data object can also be specified for the constant data object (array attributes cannot be specified).

The location of a constant data object cannot be specified because it is implicitly located by the machine at a position that provides optimum execution performance.

Space pointers cannot address constant data objects.

Entry Point

An entry point is a program object that references an instruction as the target of the program invocation functions.

When the program is invoked, the instruction defined in the external entry point for the program is given control.

When a subinvocation within the invocation is invoked, the instruction referenced in an internal entry point is given control.

An operand list that defines a parameter list can be specified in order to allow the passing of one or more arguments to the invocation or subinvocation.

Branch Point

A branch point is a program object that references an instruction as a possible target of a branching instruction.

Associated with the branch point is an instruction number that refers to an instruction in the instruction stream. A branch point can be referred to in a branching instruction or in an instruction definition list.

Instruction Definition List

An IDL (instruction definition list) is a program object that contains an ordered list of instruction stream references. These references can be either instruction numbers or references to branch points.

The IDL can be used as an operand to determine the target instruction in the Branch instruction or as a return list to allow a variable return target for the Call External instruction. In each case, an index value selects one of the entries in the IDL as a branch target.

Operand List

An operand list is a program object that provides a means of grouping references to data objects. These references define either an argument list to be passed to an invocation or subinvocation, or a parameter list to receive objects in an invocation or subinvocation.

An operand list that is used as an argument list passes the addressability of each data object contained in the list to the corresponding parameter in the invocation or subinvocation. The length of the list (number of entries) can be fixed or variable. A fixed-length argument list always passes the same number of arguments; a variable-length argument list can pass different numbers of arguments up to a maximum number. The number of arguments to be passed can be modified by the Set Argument List Length instruction. The contents of the operand list (the references to the data objects) cannot be changed.

An operand list that is used as a parameter list receives the addressability of a data object. This addressability is received during the invocation of the program or the subinvocation. The parameter list can require that either a fixed number or a variable number of arguments be passed. The number, if variable, can range from a specified minimum to a maximum value.

Exception Description

An exception description is a program object that defines the action for the machine to take when an exception occurs. An exception description entry must appear in the ODT in order for the program to specify that the exception is to be handled.

The exception description contains the following information:

- A list of the exception identification numbers to be monitored by this exception description.
- A compare value to further qualify the exception identification.
- The action to take when the exception occurs (for example, defer handling, call internal handler, or ignore exception).
- The target location where control is to be passed when the exception occurs. The location can be an entry point external to the containing program, an internal entry point, or a branch point (internal). This allows the exception condition to be handled either by another program or internally by the defining program.

Space Pointer Machine Object

A space pointer machine object is a program object that provides addressability for based data objects. The entry in the ODT (object definition table) for the based data object can specify a particular space pointer machine object as its base. The value in a space pointer machine object can then be changed in order to provide data object addressability to a specific location in a space.

A space pointer machine object has essentially the same operational characteristics as a space pointer data object. However, it exists only within the invocation of a program and it has no visible storage form, that is, no representational characteristics.

The location of a space pointer machine object cannot be specified because it is implicitly located by the machine at a position that provides optimum execution performance. Space pointer machine objects cannot be addressed by space pointer data objects or space pointer machine objects because space pointer machine objects are contained in internal machine storage rather than a space.

In general, space pointer machine objects provide better execution performance for addressing space data than do space pointer data objects. This is accomplished through internal performance benefits that can be realized because there is no defined storage form (representational characteristic) for space pointer machine objects.

Additionally, internal optimizations may be performed for some of the space pointer machine objects used in a program. The optimizations that may be performed are an attribute of the particular implementation of the machine.

Verification of Space Pointer Machine Objects

The optimizations performed on space pointer machine objects cause the detection of certain exception conditions to occur differently than from detection of the same conditions for space pointer data objects. These exception conditions are:

- Object destroyed
- Optimized addressability invalid
- Parameter reference violation
- Pointer does not exist


These exception conditions are detected for space pointer data objects when an operation requires access to or the setting of a value in a pointer. However, these same conditions for space pointer machine objects are not detected when an operation requires access to or the setting of a value in a pointer. Those operations that do not access the space data located by the value in a pointer do not detect these exception conditions. Some examples of these operations are pointer modification and comparison instructions. These conditions are always detected for operations that attempt to access the space data pointed to by the pointer's value. An example of this is copying space data where the pointer's value serves as the basing addressability for the data to be copied. For additional detail on these exceptions, refer to the *System/38 Functional Reference Manual*.



Optimization Priority Attribute

In general, space pointer machine objects provide better performance than do space pointer data objects. This is because space pointer machine objects do not have representational characteristics. The performance characteristics of pointers are also influenced by certain internal optimizations that may be performed for them.

Normally, optimizations are performed on pointers according to a priority established by a machine analysis of pointer usage within the program. If this machine analysis fails to properly prioritize a space pointer machine object, the optimization priority attribute allows the pointer to be given a priority attribute that overrides the normal priority established by the machine when programs are created. Through this attribute, space pointer machine objects can be specified as having high optimization priority even though the analysis performed by the machine would determine the space pointer machine object to be of low usage within the program. A specified priority value indicates that the pointer is of higher priority than those pointers with a lower value or those pointers for which no value is specified. A particular priority value can be specified for multiple pointers to indicate that they are of equal priority.



If no priority attribute is specified, the normal priority of space pointer data objects and space pointer machine objects controls the optimizations performed on them. When the priority attribute is specified, its effect depends upon the particular implementation of the machine. A positive effect can only be realized for some of the pointers having the priority attribute due to constraints on the optimizations that can be performed for a particular machine.

The priority attribute should be used with caution. The positive effects of the normal machine optimizations can potentially be negated through incorrect prioritization of pointers relative to their influence on the performance of a program.

Addressing

The following forms of addressing are defined and supported.

- ODT
- Pointer
- System object
- Space
 - Space pointers
 - Data views
 - Data pointers
- Argument and parameter
- Process
- Invocation
- Instruction

ODT ADDRESSING

All instruction operands, with the exception of immediate operands, refer to object views in the object definition table. An object view defines a set of attributes for the machine to use when the view is referenced as an instruction operand. This set of attributes includes the functional characteristics of the object as well as its functional location. The functional characteristics of the object determine how the object can be used by an instruction. The functional location of the object defines if, when, and where storage is to be allocated for the object as well as how the machine is to locate the object when it is referenced as an operand during instruction execution.

The ODT entry for a program object is, therefore, a description of the object rather than the actual object itself. The machine assumes the responsibility for binding operand references to the actual objects via the ODT entries.

POINTER ADDRESSING

Pointers enable objects to be addressed indirectly because a reference to the pointer is considered to be a reference to the object. For example, space pointers address spaces and, where necessary, act as an indirect reference for spaces on instruction operands or templates.

Pointers can be modified during the execution of a program in order to address different objects. Thus, for a given set of instructions in a program, the pointer may address a different object each time the instruction set is executed. The object addressed by the pointer can be within the program or outside the program.

Pointer Data Objects

Pointer data objects are stored in byte strings and have a length of 16 bytes. The boundary alignment for each byte string that contains a pointer is a multiple of 16. The alignment starts with the first byte in the space. The following pointer data objects are defined:

- Space pointer
- Data pointer
- System pointer
- Instruction pointer

A pointer data object that contains direct addressability to an object is considered to be resolved. A pointer data object that contains symbolic addressability to an object is considered to be unresolved.

Note: Because the format and the assignment of bit values within a pointer are subject to change at any time, the bit content of any pointer is unpredictable and must not be used as data by any program. The Materialize Pointer instruction must be used to determine the attributes of a pointer data object.

The following instructions perform functions that apply to the manipulation and testing of all pointer data objects.

- The Copy Bytes with Pointers instruction copies both data and pointers from one string of bytes to another.
- The Compare Pointer for Object Addressability instruction compares two pointers to determine whether they are addressing the same object or instruction.
- The Compare Pointer Type instruction compares the current type of pointer to a scalar comparison value.

Pointer data objects can also be set by materialization functions. For example, the Materialize Context instruction stores system pointers into the receiver space; these system pointers address the same system objects that are addressed by the context.

Space Pointer Machine Object

Space pointer machine objects are stored in internal machine storage areas; these objects have no defined representational characteristics other than their internal machine storage requirements. These internal storage requirements are dependent upon the particular implementation of the machine.

Space pointer machine objects provide the same addressability to byte strings in spaces as that described earlier under *Pointer Data Objects*.

In general, the instructions that can be used to change a space pointer machine object are the same as those used to change a space pointer data object.

The abnormal value attribute defined for space pointer data objects does not apply to a space pointer machine object because this attribute cannot be modified in a way that is not detectable by the machine.

Space Pointer

A space pointer provides addressability to a specific byte in the space associated with a system object (including a space object). This addressability can then be used as a base for one or more based data objects. The value (addressability) contained in the space pointer can be modified to provide variable addressability to the data object.

The addressability contained in a space pointer consists of two parts. Part 1 contains addressability to the space associated with an object (logically the first byte in the space); part 2 contains an offset into the space. This value locates a specific byte within the space.

The following instructions can be used to manipulate a space pointer:

- The Set Space Pointer instruction initializes a space pointer to address a specific byte location within the space in a system object.
- The Set Space Pointer with Displacement instruction initializes a space pointer the same as the Set Space Pointer instruction, but the resultant offset value in the pointer is modified by a signed binary value (the displacement).

- The Set Space Pointer Offset instruction assigns a value only to the offset portion of the space pointer. The space pointer addresses the same space.
- The Add Space Pointer instruction adds a signed binary value to the offset portion of a space pointer and stores the value in the receiving space pointer.
- The Subtract Space Pointer instruction subtracts a signed binary value from the offset portion of a space pointer and stores the value in the receiving space pointer.
- The Store Space Pointer Offset instruction assigns the offset value of the space pointer to a binary scalar.
- The Set Space Pointer from Pointer instruction initializes a space pointer to an address in a space based on the addressability contained in a system pointer, a data pointer, or another space pointer.
- The Set System Pointer from Pointer instruction initializes a system pointer to address the system object that contains the space addressed by any pointer.

Abnormal Value Attribute

Data objects used in more than one System/38 instruction can all receive their addressability from the same space pointer. The machine optimizes the fetching of the pointer by using the value fetched during the execution of one instruction to locate the objects referenced by subsequent instructions. The machine also monitors the pointer for modification during the execution of these instructions. When modification of the pointer is detected, the machine once again fetches the pointer to get the updated addressability.

It is possible, though, to modify the contents of a pointer in such a way that the machine does not detect the change. For example, undetected modification to a pointer occurs when more than one view of the pointer is used and the pointer is changed through one of the views. When this occurs, the updated addressability is not fetched and the desired results might not be achieved.

To prevent the situation just described, the abnormal value attribute can be specified on the pointer. This attribute tells the machine that undetected modification situations might exist and that the base addressability must be fetched each time the pointer is referenced.

A based pointer can be modified asynchronously by event handlers, exception handlers, other processes, or source/sink operations. In these situations, even though the abnormal value attribute is specified, the new addressability might not be used until the next instruction is executed. The abnormal value attribute only assures that addressability is not assumed to be the same (unchanged) from one instruction to another.

Data Pointer

A data pointer provides addressability to a specific byte in a space and provides scalar data object attributes such that the data pointer may be referenced as a scalar element.

A data pointer can address any byte in a space and can also contain any set of attributes that can be specified for a scalar data object.

A data pointer can be assigned a value as follows:

- The Set Data Pointer instruction assigns the scalar data element attributes of the source scalar object or the data pointer operand to the attribute portion of the data pointer. Either the space address of the referenced scalar object or the space address contained in the source data pointer is assigned to the target data pointer.
- The Set Data Pointer Addressability instruction assigns to the space addressability portion of the receiver data pointer a value equal to the space address of a source scalar object or the space address contained in a source data pointer. The attributes of the receiver data pointer remain unchanged.
- The Set Data Pointer Attributes instruction assigns to the attributes portion of the receiver data pointer a set of scalar data element attributes based on the value of an attribute template.
- The name resolution function enables a data pointer to be set to address an external scalar object located in the activation entry for a program.

System Pointer

A system pointer is used to address system objects. A reference to a system pointer as an operand or as an element of an attribute template is considered to be an indirect reference to the system object addressed by the system pointer.

For more information about setting system pointers and referencing system pointers, refer to *System Object Addressing* later in this chapter.

A system pointer can also contain object authorities. The authority stored in the pointer in addition to the authority available to the process through the governing user profile(s) is the total authority available to the process. The authority attribute is explicitly set by the Resolve System Pointer instruction.

Instruction Pointer

An instruction pointer is used to indirectly address an instruction in a program. The value of an instruction pointer can be modified with a Set Instruction Pointer instruction. An instruction pointer can be referenced as a branch operand to provide for variable branch targets.

SYSTEM OBJECT ADDRESSING

System object addressing locates and references system objects. The following addressing characteristics are common to all system objects:

- System objects can be known to the machine by their symbolic address. The symbolic address consists of the object's name, type, and subtype. This symbolic address is used to locate the object in the machine.
- System objects may be addressed either by one context or by no context. If a context contains addressability to the object, it is by the object's symbolic address (type, subtype, and name).
- System pointers may contain addressability only to system objects. All operand references and template references to system objects are through system pointers that address the system object. Once addressability to a system object is set into a system pointer, a reference to the pointer is an indirect reference to the object.

Symbolic Address

A symbolic address for an object consists of the object's name, type, and subtype. The symbolic address of an object can be used to locate the object through the system object address resolution functions.

The symbolic address of an object is modified by the Rename Object instruction. The object name, object subtype or both the object name and subtype can be modified by this instruction. The object is then known by the new symbolic address in the context that currently addresses that object.

System Pointer Addressing

A system pointer is used to address system objects. It provides a reference to a system object when the pointer is used as an instruction operand or the pointer is contained in an attribute template.

Addressability is established in the system pointer when:

- A system object is created. A system pointer is referenced as a receiver operand in the Create instruction; once the object is created, addressability to that object is returned in the system pointer.
- The system object address resolution functions locate a named system object and place addressability to it in the system pointer.
- A value is copied from another system pointer.
- A machine instruction (for example, one of the materialize instructions) returns addressability to a system object in the system pointer.

System Object Address Resolution

The address resolution functions cause the entries of one or more contexts to be searched in order to locate a specific system object. Once the proper entry is located, the system pointer, acting as the receiver of the address resolution function, is assigned addressability to the system object.

The system object to be located through the address resolution functions is designated by a qualified symbolic address. A qualified symbolic address consists of:

- The symbolic address of the context to be used to locate the system object. The context can be implicitly specified based on either the object type or the contents of the current name resolution list.
- The symbolic address (type, subtype, and name) of the object to be located.
- A minimum authorization qualifier for the object. This specifies a minimum authorization that must be held by a user before address resolution is performed.

Address Resolution Functions

System object address resolution can occur either explicitly based on user request or implicitly based on a reference to an unresolved system pointer.

Explicit Address Resolution: The Resolve System Pointer instruction explicitly causes system object address resolution. In this instruction, the system pointer is resolved based on the specified symbolic address. Optionally, the requested private object authorities available to the current governing user profiles can be stored into the system pointer.

The symbolic address can be specified as an operand of the instruction or as an initial value of the system pointer.

Implicit Address Resolution: If a system pointer is referenced as a source operand and is not resolved, the machine implicitly attempts to resolve addressability to a system object. If an initial value was declared for the system pointer and the system pointer has never been resolved, the initial value is used as a symbolic address of the system object to be located. If no system object is located, an exception is signaled.

If object addressability is resolved implicitly, addressability to the object is placed in the system pointer, and the system pointer is used in the operation.

Context Addressing

A context can be addressed through a system pointer or by its symbolic address (permanent contexts only). The symbolic address of the context consists of its name, object type (context), and subtype.

For system object address resolution functions, contexts are specified as follows:

- Implicitly based on object type

When the object type specifies a type that can be addressed only by the machine context, the machine context is searched to locate the object. If the object is not located in that context, an exception is signaled.

- Context qualifier in the symbolic address

The symbolic address of the object can be qualified by the symbolic address of the context to be searched to locate the object. The context is specified as a symbolic address in an initial value of a pointer, or as a pointer either in the NRL (name resolution list) or the Resolve System Pointer instruction.

The symbolic address of the object can be specified as the initial value of the system pointer or as an operand in the Resolve System Pointer instruction.

Only the specified context is searched. If the context does not contain addressability to the object, an exception is signaled.

- Contexts addressed in the name resolution list

If no context is identified in the symbolic address of the object (either implicitly by object type or explicitly by context symbolic address or pointer), the context(s) identified by the process NRL (name resolution list) is searched.

Each context is searched in the order specified by the NRL. If no object is found in any of the contexts specified in the NRL, an exception is signaled.

Name Resolution List

The NRL (name resolution list) specifies the contexts to be searched when a context is not specified as part of the symbolic address of an object and the object is not of the type addressed only by the machine context.

Whenever the Initiate Process instruction initiates a process, a space pointer specifies an NRL (name resolution list). This space pointer points to a region in a space that contains a count and a vector of system pointers to contexts. The count indicates the number of system pointers to contexts contained in the NRL. (The format of the NRL is described under the Initiate Process instruction.) The contexts are searched in the same order as they are specified in the vector. The NRL can be dynamically altered, not only by the process that contains the NRL but by another process, as long as the count is always updated to indicate that all pointers in the NRL are valid entries.

Note: All system pointers in the NRL must be previously resolved when the NRL is used for address resolution; otherwise, an exception is signaled.

Object Authorization Qualifier

The symbolic address of an object used for address resolution may be qualified by a minimum authority requirement. The minimum authorization can specify that addressability is to be returned only if the user profiles governing execution have at least the required authorization states.

If no authorization is required, addressability is returned to the first object located of proper type, subtype, and name. If authorization is required, when an object is located of proper type, subtype, and name but the required authorization states are not authorized, the search continues for an object with the proper type, subtype, name, and authority.

SPACE ADDRESSING

A space contains a set of bytes that can be referenced singly or in groups through operand references to data objects. Data objects provide the definition of scalars and pointers to be used when referencing a string of bytes as well as specifying the functional location of these bytes within a space.

Space pointers provide addressability to individual bytes within a space. However, space pointers have no scalar or array attributes. Nevertheless, space pointers can be associated with scalar data objects by the based mapping attribute. Then the scalar or pointer attributes of the data object and the addressability contained in the space pointer provide variable addressing in a space for scalars and pointers.

Data Object Addressing

A data object is a program object that provides operational and possibly representational characteristics to byte strings in spaces for use as instruction operands. A data object is defined in the ODT by a data view. It provides a logical mapping of a data view onto a set of bytes in the space. This section discusses the functional location of the set of bytes mapped to by the data view.

The data view location attribute defines the functional location of the data object. A data object may be located in a space at some particular space offset. The location of the data object is considered to be the leftmost byte of the bytes mapped to by the data view. For example, for a BINARY(4) scalar data object described to be located at offset 100 in a space, the data view maps to bytes 100 through 103 of the space.

The data view attributes that provide the functional location of a data object in a space are:

- Direct on static—which is mapped to a particular offset in the allocation of static storage for a program activation.
- Direct on automatic—which is mapped to a particular offset in the allocation of automatic storage for a program invocation.
- Based—which is mapped to a particular location relative to the space address contained in a space pointer.
- Defined—which is mapped relative to another data object.
- Parameter—which is mapped to a data object that is passed as an argument from an invoking program.

Direct Data Views

Direct data views provide a mapping relative to the static or automatic allocation in a space. The functional location of the data object is dependent on the allocation of static storage from the process static storage area, or on the allocation of automatic storage from the process automatic storage area. A data view mapped direct on automatic storage, for example, causes the location of the data object to be determined relative to the allocation of the automatic space.

The location of the data object within the allocation is determined by the position attribute, the boundary attribute, or the machine default position.

If a position value is specified, the data object is located at that position (byte) in the allocation. The first byte in the allocation is position 1.

For example:

```
C DIRECT STATIC POSITION(126) BINARY(2)
ELEMENT
```

C defines a BINARY(2) view of bytes 126 and 127 of the allocation of static storage for the program's activation entry in the process static storage area.

If the position attribute is not specified, the machine attempts to locate the data object at the next available space location. If a byte boundary is specified or if no boundary is specified, the data object is assigned to a location that is 1 byte beyond the highest assigned location. If any other boundary alignment requirement other than byte is specified, the machine assigns the data object to the next highest starting location that meets the boundary requirement. Bytes are allocated from the starting location (specified or computed) for the length specified for the object.

For example:

```
C DIRECT STATIC POSITION(126) BINARY(2)
ELEMENT
D DIRECT STATIC CHARACTER(4) ELEMENT
```

Because no position is specified for D, the machine assigns D to bytes 128 through 131 of the static allocation.

Based Data Views

Based data views provide a mapping relative to the addressability contained in a space pointer. The value of the space pointer, then, defines the functional location of the data object. This value in the space pointer can be modified to provide a different functional location for the data object.

The space pointer on which the data object is to be based can optionally be declared in the based data view. This declaration is for the duration of the program (unless overridden for a single operand using the explicit base compound operand). If a base is not declared, the based data view must be referenced in an explicit base compound operand where a space pointer is specified as a base for a single operand.

A position attribute can be specified that locates the based data object at some offset from the location addressed by the space pointer. If no position is specified, the machine assumes the location addressed by the space pointer.

Consider the following declarations:

```
X DIRECT AUTOMATIC CHARACTER(100)
  POSITION(50)
P SPACE POINTER
Z BASED(P) CHARACTER(18)
```

Executing the following Set Space Pointer instruction causes P to be set to address byte 50 of the invocation's automatic storage area. Z then will address bytes 50 through 67 of the invocation's automatic storage area.

```
SETSPP P,X
```

Executing the following Add Space Pointer instruction causes Z to address bytes 65 through 82 of the invocation's automatic storage area.

```
ADDSPP P,P,15
```

Defined Data Views

Defined data views provide a mapping relative to a previously specified data view. The functional location of the defined data object is dependent on the location of the base data object. The base data view may have a direct, defined, parameter, or based location attribute.

A position attribute locates the defined data object at some offset from the starting location of the base data object. If no position is specified, the machine assumes the starting location of the base data object.

The following example illustrates defined data views:

```
C DIRECT STATIC POSITION(10) CHARACTER(4)
P SPACE POINTER
D BASED (P) BINARY(4)
E DEFINED (C) BINARY(4)
F DEFINED (E) POSITION(3) BINARY(2)
G DEFINED (D) CHARACTER(4)
```

A reference to E refers to the same byte string as C (starting at position 10 in the static allocation) but with a BINARY(4) view.

A reference to F refers to bytes 3 and 4 of E, giving a BINARY(2) view of those two bytes. This, in turn, maps to positions 12 and 13 in the static storage area.

A reference to G refers to the same byte string as D (which is based on P) but with a CHARACTER(4) view.

Parameter Data Views

Parameter data views provide a mapping that corresponds to an argument passed on a Call External, Call Internal, Transfer Control, or Initiate Process instruction.

Scalar data objects and pointer data objects can be passed as arguments and be received as parameter data objects. When a data object is passed, the functional location of the parameter data object is the same as the argument data object. The parameter data view defines the operational characteristics to be applied to that functional location. That is, no type checking is performed to ensure that scalar data objects are passed to corresponding scalar objects of the same type, or that pointer data objects are passed to pointer data objects. The function location of the corresponding argument is used, and no position value can be specified.

Data Object Address Resolution

The data object address resolution functions enable a scalar data object with the external attribute to be located by name and have its addressability and attributes placed in a data pointer.

External Data Objects

Scalar objects that are located in static spaces and are not defined as arrays can be declared as external. These external scalar objects can then be known outside the program, but within the same process, where the scalar object is declared.

Each external scalar object must be assigned a name. This name, along with the name of the containing program, defines the symbolic address of the object.

When a program is activated, static storage is allocated and the external scalar object is assigned addressability. As long as the activation exists, the external scalar object can be located through the data object address resolution functions.

Data Pointer

Data pointers can be resolved to an external scalar data object. When resolution occurs, the space addressability and the scalar data object attributes (type and length) of the external scalar object are assigned to the data pointer. A subsequent reference to the data pointer causes the bytes of the external scalar data object to be referenced or manipulated as though the scalar data object had been referenced.

The data pointer, then, assumes the attributes of the external data object. For example, a data pointer resolved to a binary scalar data object takes on the characteristics of a binary object when referenced as a scalar instruction operand. The machine performs the indicated operation based on the current attributes and space addressability contained in the data pointer.

The attributes and addressability in data pointers can also be directly set.

Data Object Address Resolution

Data object address resolution causes the external scalar data object to be located by its symbolic address in some program activation within the process. The addressability and scalar data object attributes of the object are also stored in a data pointer. Data object address resolution can occur as follows.

- Resolve Data Pointer instruction

The Resolve Data Pointer instruction causes an external scalar data object to be located in an activation of a program in the process and also causes its addressability and its attributes to be stored in the data pointer.

The external data object is located by its symbolic address that consists of a 32-byte data object name. The name can be qualified by a reference to a program.

If a program qualifier is specified, only its associated activation is searched. Otherwise, all activation entries in the process, from the most recently activated to the least recently activated are searched until the external scalar data object is either located, or if the object is not located, an exception is signaled. The activation of the program executing the instruction also participates in the search.

The program qualifier can be specified as a system pointer operand of the instruction or through a symbolic name as an initial value of the data pointer.

Some cases when the symbolic address is not specified as an operand are as follows:

- If the data pointer has never been resolved but has an initial value declaration, the Resolve Data Pointer instruction resolves the data pointer to the external data object described by the initial value. If no data object is located, the machine signals an exception.
 - If the specified storage area does not contain a data pointer, the machine signals an exception.
 - If the data pointer is currently addressing an existing scalar object, the Resolve Data Pointer instruction causes no operation, and no exception is signaled.
- Implicit address resolution
 - When a data pointer is referenced as a source operand and the data pointer has never been resolved, the machine implicitly attempts to resolve addressability to an external scalar data object.
 - The initial value declared for the data pointer is used as a symbolic address of the external scalar data object to be located. If no data object is located, an exception is signaled.

If the data pointer is implicitly resolved, addressability to the object and the attributes of the object are placed in the data pointer, and the data pointer is used in the operation.

Array Addressing

An array is a group of data elements that have the same attributes. Scalar and pointer data views can define arrays. Each element of the array is addressable by the subscript compound operand form where a variable scalar operand or a pointer is allowed.

The subscript compound operand form consists of a primary operand (base) and a secondary operand (index) as follows:

- A base array.
- An index value that selects a specific element of the base array. The first element in the array has an index value of 1.

Arrays can optionally be constrained through use of the array constraint attribute on the create program template. This means that a range exception is signaled when a referenced element is outside the bounds of the declared array. When arrays are unconstrained, a referenced element is assumed to be within the defined bounds of the array.

A reference to an element that is outside an unconstrained array yields an unpredictable result. A range exception might be signaled depending on how certain internal functions of the machine have been implemented.

Note: An improvement in performance might be realized if the array constraint attribute is not specified.

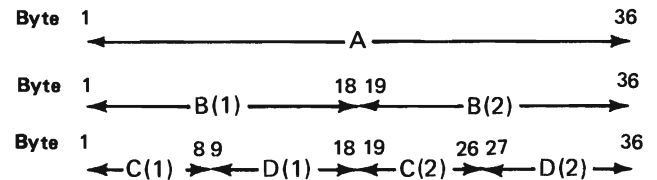
The index value must be positive. The index secondary operand can be immediate, constant, or variable. However, the index secondary operand itself cannot be a compound operand.

Individual elements of an array can be contiguous or noncontiguous. That is, an element is adjacent to the next element in the array (contiguous), or there is a uniform spacing between elements that is not the same as the length of an individual element (noncontiguous). For noncontiguous arrays, the array element offset attribute defines the number of bytes from the beginning of one element to the beginning of the next element.

For example, consider the following declarations:

```
A CHARACTER(36)
B (2) CHARACTER(18) DEFINED (A)
C (2) ZONED(8) ARRAY ELEMENT OFFSET(18)
  DEFINED (A)
D (2) CHARACTER(10) ARRAY ELEMENT
  OFFSET(18) DEFINED (A) POSITION(9)
```

The following mapping occurs:



In the previous example, A, B(1), B(2), D(1), and D(2) can be referenced as scalar operands where character strings are allowed, and C(1) and C(2) could be used as scalar operands where zoned decimal operands are allowed.

Substring Addressing

A substring is a scalar operand that addresses a portion of a declared string. A substring of the string of bytes mapped to by a character data view or a data pointer can be defined as a scalar operand by using the substring compound operand form.

The substring compound operand consists of a primary operand (base) and two secondary operands (index and length) as follows:

- A base character string.
- An index value that specifies the beginning position (in bytes) within the string. The first byte in the base string has an index value of 1.
- A length (in bytes) for the substring operand.

As a program attribute, a substring can be optionally constrained. This means that the index and length entries must be positive and must not locate any bytes outside the base string; otherwise, an exception is signaled. The index and length secondary operands can be immediate, constant, or variable, but they cannot be compound operands. When substrings are unconstrained, a referenced substring is assumed to locate bytes only inside the base string. A reference to a substring that locates bytes outside the base string yields an unpredictable result. The machine might signal a range exception in this case depending on the particular implementation of the machine.

Note: An improvement in performance might be realized if the byte string constraint attribute is not specified.

Consider the following declaration:

```
X CHARACTER(100)
```

The substring compound operand X(10,7) references bytes 10 through 16 of X. The substring compound operand X(I,7) references 7 bytes beginning at position I in X. The current value of I is determined when the instruction containing the substring compound operand is executed.

The substring compound operand X(I,J) defines a substring where the position (I) and the length (J) are determined at the time the instruction is executed.

A substring compound operand can optionally allow or disallow references to a substring with a length value of zero (null substring).

Null substring references are supported on only a subset of the instructions that support character data as operands. To determine if a particular instruction provides this support, refer to the instruction definitions in the *System/38 Functional Reference Manual*.

Space Extent Checking

Once addressability to a space has been established, the machine supports certain offset calculations within the space. For example, the Add Space Pointer and Subtract Space Pointer instructions can manipulate the offset portion of the space address contained in the space pointer.

The machine prohibits the calculation of a space offset that cannot exist. That is, neither a negative offset nor an offset larger than the maximum size to which the space can be allocated can be calculated for a space pointer.

The machine additionally prohibits a variably addressed scalar or pointer operand from referencing outside the range of the space. That is, a reference to a data object based on a space pointer causes a space addressing violation exception if the addressability of the data object is beyond the end of the allocated space (even though the space pointer contains a valid space address). A subscript compound operand or a substring compound operand can also cause the same exception if all or part of the element to be addressed is beyond the end of the allocated space.

ARGUMENT AND PARAMETER ADDRESSING

The function of arguments and parameters is to provide a means to communicate information between two execution units. The information can be transferred from an initiating process to an initiated process, a calling invocation to a called invocation, an invocation to a subinvocation, or from a subinvocation to a subinvocation in the same invocation.

The input to a program can vary from one execution of the program to the next. Because of this variance, parameters and arguments are used to provide a simple and uniform method of referring to this input or output data.

A parameter is a data object that is mapped onto an argument at the time of invocation. A parameter is not bound prior to invocation and is an indirect reference to a data object provided by the invoking program.

The parameters associated with an entry point are referenced in a program object called an operand list (used as a parameter list). The operand list is referenced by an entry point definition.

An argument is a data object that is similar to any other data object, except that it is specified in an operand list (used as an argument list).

The operand list (used as an argument list) is a set of references to other data objects that are to be passed as arguments. It is the operand of a Call External, Transfer Control, Call Internal, or Initiate Process instruction and the mechanism whereby arguments (references to actual objects) are mapped to parameters (indirect references).

During invocation, the operand list being referenced by the invoking instruction is matched with a corresponding operand list being used as a parameter list at the called entry point. For the duration of the invocation, references to the parameters in the invoked programming code refer to the corresponding arguments in the calling invocation.

Arguments

Arguments are those data objects that can be used to communicate between invocations. Arguments specify input values to the called invocation or receive output values as a result of executing the called invocation.

Passing an argument implies that the addressability to the data object is available to the succeeding invocation. Both scalar and pointer data objects can be passed as arguments.

There is no special attribute to cause an object to be considered as an argument; a reference to a data object in an argument list causes that data object to be an argument.

Parameters

Parameters are those data objects that the program receives when it is invoked. Parameter data objects allow indirect references to argument data objects passed during invocation. Addressability to the argument data object is available through a reference to the parameter data object.

A reference to a parameter object is actually a reference to the argument. Because of this implied reference, parameters are given a special parameter attribute in the ODT (object definition table). This specifies the following:

- No storage is allocated for the parameter object when the program is invoked. A reference to the parameter gains addressability to the corresponding argument as defined at the time of the invocation.
- A reference to the parameter as a source operand causes the value of the corresponding argument to be used. A reference to a parameter as a target operand causes the value of the corresponding argument to be modified.

Both the scalar and pointer data objects can be defined as parameters.

Argument Lists

An argument list (an operand list with a type of argument) is an ordered list that defines all of the arguments to be passed to a succeeding invocation on a Call External, Call Internal, Transfer Control, or Initiate Process instruction.

The argument list can cause a fixed or variable number of arguments to be communicated. A fixed-length argument list causes all of the data objects defined in the list to be passed to the succeeding invocation. A variable-length argument list allows the invoking program to determine the number of arguments to be passed from the list.

The Set Argument List Length instruction changes the number of arguments to be passed. If the instruction specifies that a definite number of arguments are to be passed, a reference to that argument list on a Call External or Transfer Control instruction passes only that number of arguments in the list. Argument lists referenced by the Call Internal instruction must be fixed-length.

An initial length must be specified in the argument list definition. This length remains in effect until it is modified. Likewise, once the length is modified, it remains in effect until further modification. The modified length can be from zero through the maximum size of the argument list.

The argument list definition must contain a reference to a data view for each possible data element in the list. These argument list references remain in effect for the life of the program. The only change that can occur in the argument list is in the number of these references that are to be communicated to a succeeding invocation or in the values of the actual data elements.

An argument list entry can reference any scalar data objects or pointer data objects. Any reference to a data object with a based, defined, direct, static, automatic, or parameter attribute is allowed; however, a reference to an array object or a substring is not allowed.

Multiple argument list entries can reference the same data view.

Addressability to an argument is established in the argument list at the time the instruction referencing the argument list is executed. The address of a variably addressed argument can be changed prior to invocation but not by the called invocation. For example, if a based data object and its base (a space pointer) were both passed as arguments, changing the value of the space pointer in the called invocation would not change the addressability of the parameter data object as known to the called invocation.

Parameter Lists

A parameter list (an operand list with the type of parameter) is an ordered list that defines all the parameters that are to be received from the preceding invocation.

The length of a parameter list can be fixed or variable. A fixed-length parameter list specifies that the entry point expects to receive exactly the number of parameters defined. Otherwise, an exception is signaled. A variable-length parameter list specifies that the external entry point expects to receive at least a minimum number of parameters but no more than the maximum number of parameters defined.

A fixed-length parameter list can receive arguments from a fixed-length or a variable-length argument list until the actual number of arguments is equal to the number the parameter list is expecting. Similarly, a variable-length parameter list can receive arguments from a fixed-length or variable-length argument list (providing that the actual number of arguments is within the dimensions allowed for the variable-length parameter list).

Parameter lists can be defined as internal or external. Internal parameter lists are referenced by internal entry points, and the lists must be fixed in length.

The parameter list definition must contain a reference to a data view for each possible data element in the list. Each data view that is referenced must have the parameter attribute. Two entries in the same parameter list cannot reference the same parameter data view. A parameter cannot be referenced in more than one parameter list (internal or external).

Argument/Parameter Correspondence

When an argument list is specified on an instruction, the individual arguments are intended to correspond one for one with the parameters in the parameter list. This correspondence includes the number of arguments and parameters (as stated under *Argument Lists* and *Parameter Lists*); however, the machine does not verify that the parameter and the argument are of the same type.

Figure 2-1 shows the relationships between argument-parameter binding. This figure shows the high-level language form and the machine interface program template form of two programs, X and Y.

When program Y is invoked by program X, parameters A, B, and C are bound to corresponding arguments U, V, and W. The Add Numeric instruction adds the values of V and W and stores the result in U.

High-Level Language Form	
Program X	Program Y
PROC	PROC(A,B,C)
DCL U	DCL A PARM
DCL V	DCL B PARM
DCL W	DCL C PARM
.	.
.	.
CALL Y(U,V,W)	A = B+C
.	.
.	.
END X	END Y

Program X

Object Definition Table	Instruction Stream
Entry Point X Instruction 1 External	•
U Scalar Data Object	•
V Scalar Data Object	CALL E,Z
W Scalar Data Object	•
E System Pointer Initialize Program Y	•
Z Operand List Argument List (Fixed,3,U,V,W)	•

Program Y

Object Definition Table	Instruction Stream
Entry Point Y Instruction 1 External Operand List (D)	•
A Scalar Data Object Parameter	•
B Scalar Data Object Parameter	ADDN A,B,C
C Scalar Data Object Parameter	•
D Operand List-Parameter List (Fixed,3,A,B,C)	•

Figure 2-1. Argument/Parameter Relationship

PROCESS ADDRESSING

Processes are identified by a reference to the process control space associated with the process.

INSTRUCTION ADDRESSING

Program instructions are numbered sequentially beginning with the value one for the first instruction, two for the second, and so on. This sequential order is the basis for addressing instructions through the ODT and in branching instructions.

Instruction Numbers

Instruction numbers provide the most basic form for addressing instructions. Instruction numbers can be used as immediate branch targets, indirect branch point references, return point references, instruction pointer references, and instruction definition lists.

Immediate instruction numbers can be absolute or relative. Absolute instruction numbers refer to the sequential number of the instruction relative to the beginning of the instruction stream. Relative instruction numbers refer to a sequential number of the instruction relative (positive or negative) to the instruction where the reference is made (relative instruction 0). All instruction stream references must be within the range of the instructions defined in the instruction stream of the program template.

Branch points are defined in the ODT and refer to absolute instruction numbers. Branch points can be referenced as branch targets.

Instruction Pointers

Instruction pointers provide variable addressability to instructions within a program. An instruction pointer can be set by the Set Instruction Pointer instruction with a branch target ODT entry as an operand or with an immediate value (absolute instruction number or relative instruction number). The actual value in the pointer is the number of a machine interface instruction. An instruction pointer can be specified as the target of any branching instruction. In a branch instruction, a reference to the instruction pointer results in a branch to the instruction to which the pointer is set. An instruction pointer can address an instruction only in the program in which it is set, and an instruction pointer can be used as a branch operand only in that program.

Instruction Definition Lists

An IDL (instruction definition list) defines a set of instruction numbers and refers to branch points. An IDL is specified in the ODT and can be referred to by the instruction stream but not modified. Instructions referred to in an IDL must be within the range of the program instruction stream in which the IDL is used (IDLs cannot be referred to outside a program).

An IDL can be used in a compound operand of a branch operand. The index suboperand furnishes an integer value that is used as an index into the IDL to determine the branch target (one for the first element of the IDL, two for the second, and so on).

For example, with the following definitions:

```
L1   Branch Point
L2   Branch Point
IDL1 IDL (5,L1,L2)
```

An IDL consisting of an instruction number (5) and two branch points (L1 and L2).

the following Copy Numeric Value and Branch instructions cause a branch to the third element of the IDL, which is set to label L2.

```
DCL   C BINARY(2)
CPYV  C,3
B     IDL(C)
```

In a similar manner, a return list can be defined for a Call External instruction. The return list is an IDL and allows the caller to specify the list of return targets allowed based on certain situations that may be detected in the called program or subprogram. The called program can select a return target based on an index value specified on the Return External instruction. If no value is specified, control returns to the instruction following the Call External instruction. Otherwise, a target is selected from the return list and control is passed to that instruction. The ordering of return targets and of conditions detected is dependent on user conventions.

Context Management

A context is a system object that contains addressability by symbolic address to other system objects. Addressability to a system object can be placed in a context, and the context can then be used to locate the system object by its symbolic address. A temporary context is not inserted in the machine context and therefore cannot be symbolically addressed. The symbolic address of an object used for context addressing is by type, subtype, and name. The symbolic addresses of all objects addressed by a single context must be unique. For example, two objects named ABC of the same type and subtype cannot be addressed by the same context. A maximum of one context can be used to address a system object.

KINDS OF CONTEXTS

There are two kinds of contexts in the machine: the machine context and user-defined contexts.

Machine Context

The machine context is implicitly created and maintained by the machine. It contains addressability to the following types of objects: user profiles, contexts, logical unit descriptions, controller descriptions, and network descriptions. When an object of this type is created, addressability to the object is implicitly inserted into the machine context. No other context can address these types of objects as they are always addressed by the machine context. The user can never obtain a system pointer to the machine context.

User-Defined Context

A user-defined context is created explicitly through use of the Create Context instruction. This kind of context can address any system object except those that are restricted to the machine context or those system objects already addressed by a context (only one context can address a system object). Permanent and temporary system objects can be addressed by permanent or temporary contexts.

CONTEXT MANAGEMENT FUNCTIONS

Context management functions are defined as context creation, context destruction, and addressability modification.

- Context creation—Contexts are created by the Create Context instruction. The machine context is implicitly created by the machine.
- Context destruction—Contexts are destroyed by the Destroy Context instruction.
- Addressability modification—Addressability can either be implicitly inserted into a context when the object is created or be explicitly inserted into a context by the Modify Addressability instruction. Addressability to an object is implicitly deleted from a context when the object is destroyed, or it may be deleted from a context by the Modify Addressability instruction.

Objects addressable by the machine context have addressability inserted into that context when they are created and deleted when they are destroyed; these objects cannot be addressed by any other contexts.

The Create instruction for system objects can specify a context where addressability is to be inserted when the object is created. If no context is specified, addressability is not placed in any context unless the object is of the type that must be addressed by the machine context.

Object addressability contained in a context is implicitly deleted when the object is destroyed.

The Modify Addressability instruction causes the object addressability contained in a system pointer to be inserted into or transferred to a specific context (if specified) and removed from the context currently addressing the object (if any).

Object addressability contained in a specific context for a specific object is explicitly deleted when a Modify Addressability instruction is executed. Addressability cannot be deleted if the object is addressed by the machine context.

Materializing Contexts

The current contents of a context can be materialized. All of the context entries or some subset of the context entries can be materialized by the Materialize Context instruction. If only a subset of the context entries is materialized, the selection can be based on type, type and subtype, name, type and name, or type, subtype, and name.

All entries that satisfy the search criteria are materialized (for example, all queues in the context). When an object name is specified as the search criteria, the number of characters in the context entry is compared to the number of characters in the name (the compare operation starts with the leftmost character). Therefore, entries for objects AB, AC, ABC, and ACB can be materialized with a search criteria of A.

For each object selected, a system pointer addressing the object, the object's symbolic address, or both, can be materialized.

CONTEXT AUTHORIZATIONS

The contents of a context can be controlled by the following object authorizations:

Authorization	Operations
Retrieve	Materializing context. Resolving addressability through a context.
Insert	Creating object with initial context addressability. Context receiving addressability via the Modify Addressability instruction.
Delete	Context losing addressability via the Modify Addressability instruction. Destroying an object does not require delete authority to the addressing context.
Update	Rename Object instruction.

Note: There is no authority requirement for the operations involving the machine context.

Authorization Management

Authorization management controls the use of objects, resources, certain instructions, and various machine attributes. System/38 provides this control by:

- Preventing some users access to an object while permitting other users restricted or complete use of the object.
- Limiting the quantity of a resource that a user is allowed to use.
- Selecting the users that can issue certain instructions. A user can be given the authorization to any combination of these instructions.
- Selecting the users that can modify certain machine attributes. This authorization is given by the machine attribute group. A user can be given the authorization to modify machine attributes in any combination of these groups.
- Selecting the users that can control the use of the machine.

USER PROFILES

The authorization to use the system resources is controlled by a system object called a user profile. Each user of the system is identified by a user profile (a user is considered to be one or more users associated with a single user profile).

User profiles are created with initial values that define a portion of the user's processing environment and are subsequently updated to include the entire scope of authorization. A user profile has a name, type, and subtype that identifies a user to the machine in one of the following manners:

- All users of the system can be identified to the system by the same user profile.
- A group of users can share a single user profile.
- Each user of the system can have a unique user profile.

Each process in the machine executes under control of a user profile. (A user profile is specified as a process attribute.) When the machine executes an instruction, references an object, or requests a resource, it is done in the name of the user. Therefore, the user profile associated with the process is checked for the authority to use that item. An authorization exception is signaled when a process attempts to exceed its authorization by requesting the use of an object, function, or resource for which its user profile is not authorized.

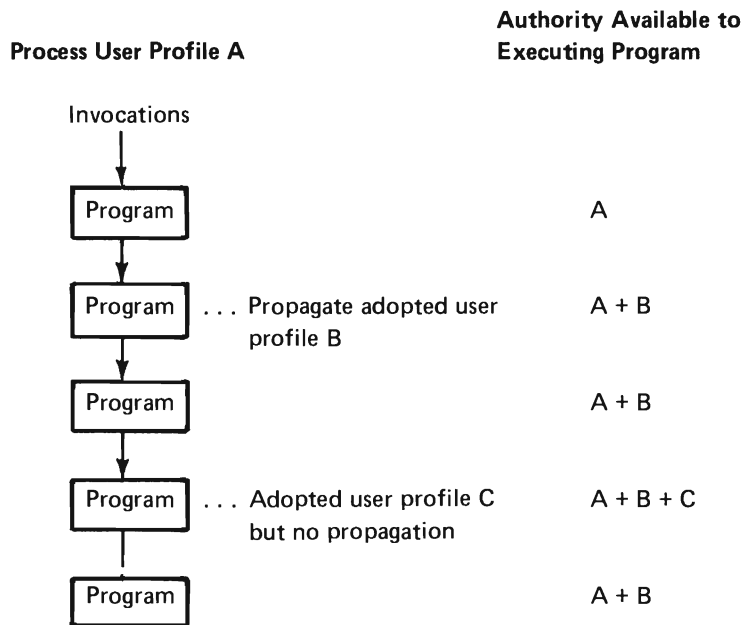
When a process creates a permanent system object, the user profile associated with the process (referred to as the process user profile) is assigned ownership of and all rights to that object. Temporary objects are not owned by any user profile because when a temporary object is created, all object authorities are granted to the public. In addition, object authorities can neither be granted to nor retracted from a temporary object.

Adopted User Profile

A process can gain additional authority via an adopted user profile. A program can have an attribute to indicate that while the program is executing, the process has the authority to use the program owner's user profile in addition to the process's user profile. In other words, the user profile that owns the program can be adopted by the process as a second source of authority.

The adopted user profile propagation program attribute indicates whether the adopted user profile authorities are to be available to programs called by the adopting program; when this attribute is specified, the adopted user profile is propagated to subsequent invocations.

The following diagram illustrates the range of authority provided by the adopted user profile with respect to invocations of other programs with or without an adopted user profile:



The adopted user profile's authority is available to succeeding invocations of other programs unless the program attribute indicates no propagation. All authorizations come from the process user profile and the adopted user profiles except for the resource authorization, which is obtained only from the process user profile.

The authorizations and/or resources associated with user profiles are:

- **Object authorizations:** A system object can be collectively or individually authorized to users for specified types of operations.
- **Special authorizations:** Special authorizations allow certain implicit authorities that are not necessarily associated with one specific instruction or object.
- **Resource authorizations:** Resource authorizations control the amount of system resource that the user can utilize.
- **Privileged instructions:** A user can be authorized to execute certain privileged instructions.
- **Owned objects:** Certain authorities are implied to the owner of a system object. The storage resources consumed by system objects are charged against the owner's resource authorization.

Object Authorization

Instructions that involve system objects usually require certain authorities to those objects before a process can complete the operation. During the execution of a program, instructions may require one or more of the following specific object authorities, which can be granted to users in any combination:

- **Object control**—for example, destroying an object or transferring ownership of the object.
- **Object management**—for example, grant/retract authority, or modify addressability.
- **Authorized pointer**—for setting the authority attribute in a system pointer.
- **Space**—for obtaining a space pointer addressing the object's associated space from a system pointer addressing the object.

- **Retrieve**—for example, retrieving entries in a data space or finding entries in an index.
- **Insert**—for example, inserting new entries into a data space or enqueueing messages on a queue.
- **Delete**—for example, removing entries from a data space or removing addressability from a context.
- **Update**—for modifying entries in a data space or replacing an entry in an index.

A process has four principal sources for object authorities:

- Authority stored in a system pointer.
- The object's public authorization.
- Implicit authority through all object special authority.
- Private authorization.

When a permanent system object is created, the user profile of the creating process becomes the owner of the object and is implicitly granted all of the object authorities to that object as a private authorization. The object is not authorized to any other user profile on creation.

Two instructions (Grant Authority and Retract Authority) are used to modify object authorizations. These instructions grant the object authorities to a specific user profile or to all user profiles, that is, the public.

The Grant-Like Authority instruction grants authority for all objects in a referenced user profile to another user profile.

A system object has two principal classes and recipients of authorizations: public and private. Public authorization is a general authorization supplied to every user profile without the profile being individually authorized. Private authorizations enable certain user profiles to be authorized with specific authorizations to a system object. The authorization granted to the owner is a special case of the private authorizations. The Grant Authority, Grant-Like Authority, and Retract Authority instructions require certain object authorizations to the object being authorized. The owner does not require the necessary object authorizations to an object in order to successfully grant/retract authority to or from it. The owner may desire to protect his own object from accidental modification by retracting all but the necessary authority from himself. In this way, it is unlikely that an inadvertent reference to an object during a modify function could damage that object.

For efficiency, the authority available to a user can be stored in the system pointer to the object. If the required authorization is contained in the pointer, no further authority checking is required. Storing authority in the pointer has performance benefits even though there are some functional disadvantages. For example, an authorized system pointer can be saved indefinitely. This renders the Retract Authority instruction ineffective for such cases. Furthermore, a user can pass an authorized pointer to a nonauthorized user. If this security exposure is undesirable, at the cost of efficiency the user can prevent the storing of authority in the system pointers. For this reason, one of the object authorizations governs the ability to store the authority in the pointer.

All-object authority is a special authorization that allows a user unlimited access to all objects. Wherever object authority of any kind is required, this special authority is sufficient, and no other object authorities are required.

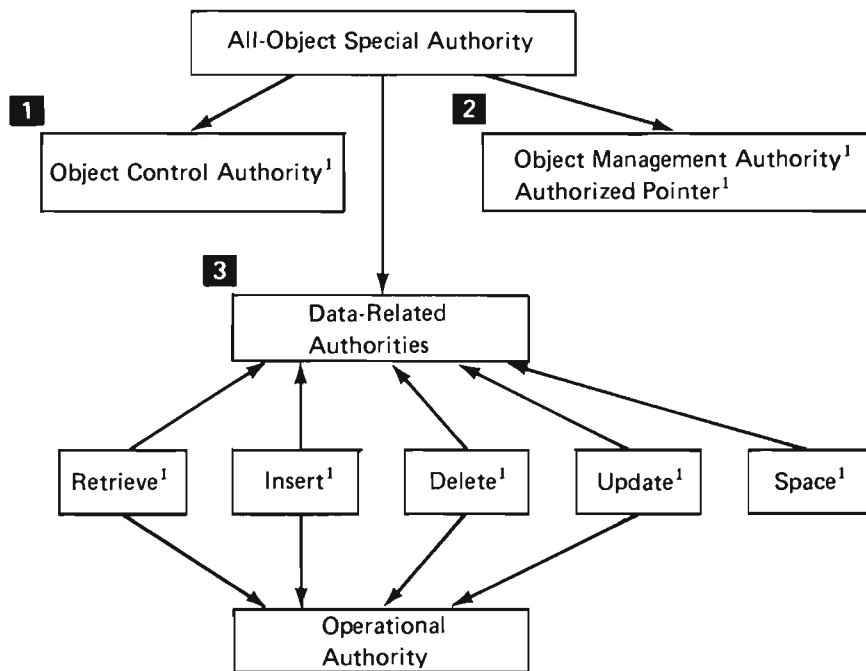
When the authorization for an object is checked, the authority is considered to be cumulative with respect to all sources of authority. For example, part of the required authority can be obtained from the object's public authority, while the remaining part can be obtained from private authorization. Each source may not have the required authorization alone, but when object authority is checked, all available object authorizations are combined. If the program being executed adopts a user profile, the adopted user profile's authority is also combined with all other sources of authority.

Object Authorization States

The object authorities are subdivided into three distinct categories (Figure 2-2):

- 1** Object control—which controls ownership and existence of objects.
- 2** Object management—which controls access and availability of objects.
- 3** Data-related authorities—which control reading, writing, and general usage of system objects.

Notice that these categories are for different purposes; neither object control nor object management authorities allow the contents of an object to be read; neither the object control nor the data-related authorities allow authority to be granted or addressability to be affected. Only the object control authorities allow the ownership or the existence of an object to be changed.



¹Object authorization which can be granted

Figure 2-2. Hierarchy of Object Authorization

In addition to being data related, the retrieve, insert, delete and update authorities are also known as operational authorities because a process has an implied operational authority when it has one or more of these authorities. Operational authority is required for operations that are not within the previous categories. For example, operations such as materializing object attributes require operational authority. Some objects do not have any instructions that require any of the four authorities but rather require operational authority. For such objects, operational authority is granted by granting all four of the operational authorities. When all four authorities are not granted for these types of objects (except a space object), an exception is signaled. The operational authorities of a space object can be used for user-defined purposes.

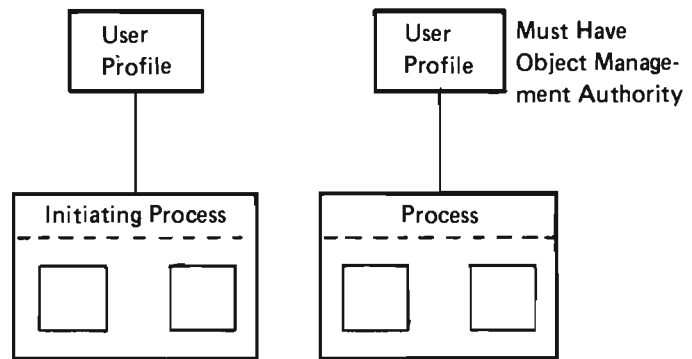
Object Control: Object control authorization is required to modify the ownership and the existence of an object. The following operations require this authority to the object:

- Destroy object—Object control authority to an object is required in order to destroy that object. However, there are no authorities required to the addressing context, the owning user profile, or the access group that contains the object to be destroyed.
- Transfer ownership—Object control authority to the object, delete authority to the old owning user profile, and insert authority to the new user profile are required to transfer ownership of an object. After the transfer operation is completed, all user profiles retain the same authorities to the object; only the ownership changes. The resource authorization on the new owner's user profile is observed when determining whether the transfer is possible. A program that adopts a user profile cannot have its ownership transferred unless the process user profile has all-object special authority. This restriction provides control over a user who might be able to transfer such a program to another user profile, which would then have authorities the transferring user was not intended to obtain. Then the user could execute the restricted program with those authorities obtained through the adopted user profile.
- Suspend object—Object control authority and restricted suspend special authority are required to suspend an object. If the process has unrestricted suspend special authority, object control is not required.
- Load object—Object control authority and restricted load special authority are required to load an object (replace an existing object). If the process has unrestricted load authority, object control authority is not required.

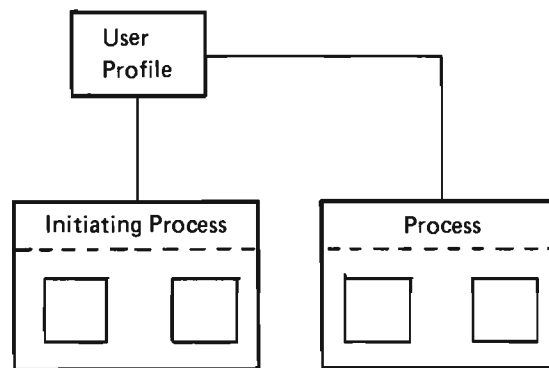
Object Management: Object management authority is required to control the accessibility and availability of system objects. The following operations require this authority:

- **Modify addressability**—Object management authority to the object, delete authority to the old addressing context, and insert authority to the new addressing context are required to change the addressability of an object. The addressability of objects addressed by the machine context cannot be modified because no authority is available to the machine context.
- **Grant authority**—Object management authority and the object authorities to be granted are required for a nonowner to grant authority to an object. Only the owner of an object, or a user profile with all-object special authority, is allowed to grant object management authority. The owner can always grant any of the object authorities.
- **Retract authority**—Object management authority and the object authorities to be retracted are required for a nonowner to retract authority to an object. The owner can always retract any of the object authorities.
- **Rename**—Object management authority to the object and update authority to the addressing context are required to rename an object.
- **Create cursor**—Object management authority is required for the data space(s) over which the cursor is being created. A cursor exercises a type of access control to data space entries because all access to a data space entry must be through a cursor. Additionally, a cursor determines the fields of a data space entry that can be operated upon.
- **Data base maintenance**—Certain data base maintenance options require object management authority to the data space or data space index.
- **Modify attributes**—Object management authority is required to modify object attributes.
- **Initiate process**—Object management authority is required for the user profile under whose control the process is to execute (unless it is the same user profile as that of the initiating process).

Different Object Management Authority



Same Object Management Authority



Authorized Pointer: The amount of system time required to check the authorizations of an object can be decreased by storing these authorizations in the system pointer. The authorization stored in the system pointer is examined when determining the process's authority to the object. Also, storing an authorization in the pointer has other functional implications. For example, pointers can be saved in spaces indefinitely and retain any authority stored in these spaces even though the authorities have been retracted by the Retract Authority instruction. Therefore, the Retract Authority instruction is rendered ineffective for the cases where authority is stored in the pointer. In addition to saving authorized pointers, a user can pass them to other users who have not been explicitly authorized to use an object. An implicit granting capability is therefore possible with authorized pointers. The passing and saving of authorized pointers may be a desirable function in some applications. For applications in which this is not desirable, the storing of authority in system pointers can be prohibited by not granting authorized pointer authority to the object. That is, a process must have authorized pointer authority to the object in order to have any authority stored in a system pointer pointing to that object.

Space: Any system object can have an associated space. (A space object is an object with nothing but an associated space.) All data in the associated space is referenced via a space pointer. Space authority allows a user (if desired) to access the object and not the space. In order to obtain a space pointer for the space associated with a system object, the process must have space authorization to that object. After the space pointer is obtained, there is no further control on the access to the space.

Retrieve: Retrieve authority is required in a wide variety of instructions. It permits retrieval, examination, or reading of the object's contents. Some examples of these contents are data space entries in a data space, entries in an index, objects addressed by a context, objects owned by a user profile, messages on a queue, members of an access group, and instructions in a program.

Insert: Insert authority is required by instructions that add new entries to objects. Some examples are inserting an entry into a data space or index, enqueueing a message on a queue, placing addressability to an object into a context, and making a user profile the owner of an object.

Delete: Delete authority is required by instructions that remove existing entries from an object. Some examples are removing a data space entry or an index entry, dequeuing a message from a queue, and removing addressability from a context.

Update: Update authority is required by instructions that modify existing entries in an object. Some examples are updating a data space entry, replacing an index entry, or renaming a context entry.

Operational: The object authorities (retrieve, insert, delete, and update) are known as operational authorities. If the process has any one or more of these operational authorities to an object, the process has operational authority. Many instructions only require operational authority because either the operation is not one that is categorized into one of the operational authorities, or the object does not support any distinction between them.

If desired, programs can be restricted to execute-only authorization. This restriction is possible because retrieve authority is required to materialize the source template of the program and only operational authority is required for execution of the program. If it is desired that users only execute the program, one or more of the operational authorities, except retrieve, should be granted.

Figure 2-3 shows the object authorities that are supported for each object.

Object Type	Object Authorities							
	Object Control	Object Mgmt	Authorized Pointer	Space	Retrieve	Insert	Delete	Update
Access group	X	X	X	X	X	X	X	X
Commit block ¹	X	X	X	X	0	0	0	0
Context	X	X	X	X	X	X	X	X
Controller description ¹	X	X	X	X	0	0	0	0
Cursor ¹	X	X	X	X	0	0	0	0
Data space	X	X	X	X	X	X	X	X
Data space index ¹	X	X	X	X	0	0	0	0
Dump space	X	X	X	X	X	X	0	0
Index	X	X	X	X	X	X	X	X
Journal port	X	X	X	X	X	X	0	0
Journal space	X	X	X	X	X	X	0	X
Logical unit description ¹	X	X	X	X	0	0	0	0
Network description ¹	X	X	X	X	0	0	0	0
Process control space ¹	X	X	X	X	0	0	0	0
Program	X	X	X	X	X	0	0	0
Queue	X	X	X	X	0	X	X	0
Space	X	X	X	X	0	0	0	0
User profile	X	X	X	X	X	X	X	X

X = Supported
0 = Not specifically supported but used for indicating operational authority

¹Operational authorities must be granted and/or retracted with all 1's (ones) or 0's (zeros).

Figure 2-3. Valid Object Authorities

Special Authorizations

Special authorizations that can be assigned to a user are implicit object authorizations, function authorizations, and machine attribute modification authorizations.

Special authorizations allow a user to perform certain special functions such as operate on objects using implied authorizations and modify machine operation. (A user in a supervisory role over machine operation might be granted such an authorization.) A user can be given any combination of the following special authorizations:

- All-object authority—This special authorization provides the user with authority to use any object in the machine without public or private object authorization being granted for the object. A user profile with this authority has no need for any other implicit or explicit object authorization. A user profile without this authority must have separate object authority to perform any function against an object. All-object authority does not imply load or dump special authorities.
- Load—This special authorization allows the user to perform the load function in order to copy an object from a load/dump medium onto the machine. This authorization has two options:
 - Load restricted—specifies that the user may execute the load function for objects that can be loaded and for which the user is the owner, has object control authority, or has all-object authority. Space authority is also required when the object has an associated space.
 - Load unrestricted—specifies that the user may execute the load function for any object that can be loaded. No object authorization is required.
- Suspend object—This special authorization allows the user to execute the Suspend Object instruction. This authorization has two options:
 - Suspend restricted—specifies that the user may execute the instruction for objects that can be suspended and for which the user has either object control authority or all-object authority.
 - Suspend unrestricted—specifies that the user may execute the instruction for any objects that can be suspended. No object authorization is required.
- Dump—This special authorization allows the user to perform the dump function in order to copy an object from the machine to a load/dump medium. This authorization has two options:
 - Dump restricted—specifies that the user may execute the dump functions, but only for objects that can be dumped and for which the user has either object management (if a data space is to be dumped), space (if the object has an associated space), and retrieve authorities, or all-object special authority.
 - Dump unrestricted—specifies that the user may execute the dump function for any object that can be dumped. No object authorization is required.
- Process control—This special authorization allows a user, other than the original initiator of a process, to perform the following operations on that process:
 - Materialize process attributes
 - Modify process attributes
 - Suspend a process
 - Resume a process
 - Terminate a process
- Service—This special authorization allows machine and I/O device maintenance operations to be initiated through the Request I/O instruction.

Nine groups of machine attributes describe the actual characteristics or status of the machine. Group 1 contains attributes that can be modified without special authority; groups 2 through 9 contain attributes that cannot be modified without special authority. To modify a machine attribute in groups 2 through 9, authority is required for the group in which the machine attribute appears. The Modify Machine Attributes instruction allows a subset of a group of machine attributes to be selected. The group is selected through the attribute section operand.

Resource Authorization

Resource authorization is used to limit the amount of auxiliary storage that can be allocated for storing permanent objects owned by a user profile.

Privileged Instructions

Most System/38 instructions are available to all users of the machine; that is, the machine does not prevent users from executing these instructions. However, the machine does limit the use of some System/38 instructions called privileged instructions.

Privileged instructions are used to restrict creation or attribute modification of certain types of system objects. Privileged instructions also limit the use of certain functions. The user profile is granted authorization to privileged instructions through use of the Create User Profile instruction or the Modify User Profile instruction.

The following are privileged instructions:

- Create Logical Unit Description
- Create Network Description
- Create Controller Description
- Create User Profile
- Modify User Profile
- Terminate Machine Processing
- Initiate Process
- Modify Resource Management Controls

A user profile that is authorized to use the Create User Profile or Modify User Profile instruction can provide to other user profiles the authority to use any of the privileged instructions for which the user profile is itself authorized.

A privileged instruction exception is signaled when an unauthorized user attempts to execute a privileged instruction.

AUTHORIZATION FUNCTIONS

Authorization functions involve making a user known to the system and having that user's authorization monitored and controlled by the machine.

Enrolling Users

New users are enrolled in the machine when a user profile is created in their behalf. A user profile is created through use of the Create User Profile instruction. The user profile owning the process that executes the instruction becomes the owner of the new user profile.

The authorizations that can be initially assigned to the new user profile are privileged instruction, special, and resource.

The user profile (which includes the process user profile and any currently adopted profiles) that is controlling the process when the new user profile is created must have authority for the Create User Profile privileged instruction. Additionally, the new user profile cannot be granted more privileged instructions or more special authorizations than owned by the creating user profile.

The only context with addressability to the user profile is the machine context. Therefore, the name and subtype specified for a user profile must be unique when compared to all other user profiles.

To remove a user profile from the machine, the Destroy User Profile instruction is executed. The user profile being destroyed cannot own any objects that currently exist except itself, and no processes can be executing under its control.



Modifying Authorization

After a user profile is created, the authorizations held by that user can be changed. The following functions change the authorizations available to the user profile.

Modify User Profile Instruction


Privileged instructions, special authorizations, and resource authorizations can be modified to add or retract the authorization(s) held by the user profile.

The Modify User Profile instruction is a privileged instruction.

Object Creation

When a permanent object is created in a process, the user profile under which the process is executing is made the owner of the object. The owner is initially granted all object authorities to the object.

Grant Authority Instruction



A user profile with object management authority for an object can grant some of its object authority for that object (except object management) to another user profile or the public. The owner can always grant any object authority for the object, even to himself. Only the owner of an object or a user profile with all-object special authority can grant object management authority.

Grant-Like Authority Instruction

This instruction grants authority from a referenced user profile to a receiving user profile. The authorities for all objects in the referenced user profile are granted. All new authority codes can be granted if the receiving user profile has no authority to the object. In addition, new authority codes can be added to authority codes previously granted.

Retract Authority Instruction

A user profile with object management authority for an object can retract its object authority for that object from another user profile or the public. The owner can always retract any object authority for the object, even from himself.


Transfer Ownership Instruction

The ownership of an object is transferred from the current owner to another user profile. All object authorizations that any user profile (including old owner) had before the transfer remain unchanged.

The user profile transferring ownership of an object must have object control authority for the object whose ownership is being transferred.

A user profile that has private authority for an object retains private authority even though ownership to the object is transferred. The new owner is given all private authorities.

Testing for Authority



The Test Authority instruction is used to test or retrieve the object authority that is currently available to the process.

Materializing Authority

The current authorization status in the system can be made available anytime during machine operation. The following instructions are used to determine this authorization.

Materialize User Profile Instruction

The Materialize User Profile instruction materializes the current authorization status of the user profile. The authorization status includes:

- Authorizations for privileged instructions
- Special authorizations
- The amount of auxiliary storage allocated and used by this user profile

Materialize Authority Instruction

The Materialize Authority instruction materializes the specific types of object authority for a system object available to a user profile. Public authority for the object can also be materialized.

Materialize Authorized Objects Instruction

The Materialize Authorized Objects instruction materializes optional information about the objects owned by the user profile, the objects privately authorized to the user profile, or both the objects owned by the user profile and the objects privately authorized to the user profile.

Materialize Authorized Users Instruction

The Materialize Authorized Users instruction materializes the list of all user profiles that have private or ownership authority to a specific system object; the public authority is also materialized. The following items are materialized for each user-listed profile:

- System pointer to the user profile
- User profile name (optional)
- User profile subtype
- Private authority

Authority Verification

During the execution of a process, the machine verifies that the process has sufficient authority to perform the following functions:

- Reference a system object
- Perform a function requiring special authorizations (for example, modify machine attributes)
- Create or extend a permanent system object
- Execute a privileged instruction

Authority verification can occur through any of the following means:

- Public authority (only for object authorizations)
- Process user profile
- Process adopted user profiles
- Adopted user profiles
- System pointer authority attribute (only for object authorizations)

Public Authority

Public authority relates to the object authority granted to all user profiles in the machine. When public authority is granted for an object, no user profile is specified; the authority specified is given to all user profiles. When the new user profiles are created, they are given authority to all objects with public authority.

All of the object authorities do not have to be granted as public authority. Public authority can be granted, for example, for operational authority, but the owner can withhold public rights to object control and object management authorities. In this example, any user profile in the machine can functionally use the object but cannot destroy the object, put addressability into another context, or grant any private authority to the object.

Process User Profile

Each process executes under control of a user profile called a process user profile. A process user profile can be used to verify all authorizations required by the process including object authorizations, special authorizations, resource authorizations, and privileged instruction authorizations.

A process user profile is specified when the process is originally initiated. The process user profile can be changed by the Modify Process Attributes instruction. The user profile of the initiating or modifying process must have object management authority for the specified user profile unless it is the same user profile. A user profile must also have process control special authorization in order to modify a process not originally initiated by the user.

A process can have its governing user profile replaced with another user profile if authority to do that was not specifically disallowed when the process was originally initiated.

Process Adopted User Profiles

A process can gain additional authority through process adopted user profiles. These process adopted user profiles are associated with the adopting process until it is terminated (unless modified by a Modify Process Attributes instruction). All authorities granted to these user profiles are available to the adopting process. The ownership of objects created by the process are always reflected in the process user profile. Event and exception handlers maintain authorities available through process adopted user profiles.

One reason for adopting user profiles on a process basis is that certain user profiles could be considered as group profiles. This means that objects created under these user profiles or authorities granted to these user profiles may be available to a large number of users without granting authority for each object to each individual user.

Program Adopted User Profiles

In addition to the authority provided by the process user profile and the process adopted user profiles, a process can also adopt other user profiles and, therefore, use any additional authority available to these user profiles.

The means available for a process to adopt other user profiles are as follows:

- When a program is created, the creator can specify that the program is to have the adopted user profile attribute.
- When the program is invoked, the user profile is adopted by the process, and the authorization rights of the owner of the program are temporarily available to the process.
- When a program is created with the adopted user profile attribute, this same program can also have the propagate adopted user profile attribute. When a program with these two attributes calls another program, its adopted user profile authorities are also available to the called program as well as any additional called programs.
- When authority is to be verified, the authority available to the adopted user profile and the authority available to the process user profile is checked.

If an invoked program or a process does not adopt a user profile, then only the process user profile is used.

Adopted user profiles allow a function to be defined by a user. This ensures that all authorizations required by the function are available. It also allows other users to be authorized to the function by authorizing them to the first program in the function. If the first program invoked by the function adopts the definer's user profile, all programs invoked for the function have the benefit of the definer's authorization.

An external event or exception handler does not receive the authorization from any previously adopted user profiles. The exception handler program may, however, adopt a user profile.

Adopted user profiles are used for all authorizations except resource authorization. Only the process user profile is used for resource authorization.

If the ownership of a program is transferred to a new owner (user profile), the user profile of the new owner is used as the adopted user profile when the program is subsequently invoked. The ownership of a program with the adopt user profile attribute cannot be transferred unless the process transferring the ownership has all-object special authority. A transferred program that has the adopt user profile attribute and is currently invoked continues to use the old owner's adopted user profile until a new invocation occurs.

System Pointer Authority Attributes

A system pointer can contain addressability to a system object as well as object authorities for using the object.

When an object is referenced through a system pointer, the authorization attributes in the system pointer can provide sufficient authority for the object to allow its use in the instruction. If the system pointer does not contain enough authority for the operation, public authorization, the process user profile, and any current adopted user profiles are used for further verification.

A user can request that the system pointer be resolved and/or the authority be stored in the system pointer by using the Resolve System Pointer instruction. This instruction can also cause the authority attribute not to be stored (meaning no authority in the system pointer).

The user controls system pointer authority. This authority is not stored by the machine unless requested by the user.

A system pointer with the authority attribute can be copied outside the process and thereby cause implicit granting of authority to another process that may use the pointer or a copy of the pointer. This function can occur when the system pointer is copied to a permanent space object, enqueued as part of a message on a queue, contained in a space object used as event-related data, passed as an argument to an initiated process, or passed to an initiated process as an element of the process communication object. The authority cannot be stored in the system pointer unless the process has authorized pointer object authorization to that object. By not granting this authority, the user can control the implicit granting of authority.

Test Authority Instruction

The Test Authority instruction verifies that the object authorities and/or ownership rights for an object are currently available to a process. The authorities and ownership rights are contained in a template that is specified as an operand on the instruction. The contents of the template are tested against the authorities currently available to the process. The results of the test provide the following options (if specified by the instruction):

- Transfer control conditionally to the instruction indicated in a branch target (branch form).
- Assign a value to each indicator operand (indicator form).

If no branch options are specified, program execution continues with the next sequential instruction.

The Test Authority instruction also returns the object authorities and/or ownership rights (as specified in the authority template) to a receiver operand (if specified).

Authority Verification Summary

Several functions can be used to verify proper authorization. The authority verification functions obtain authorization information from one of several places. The following describes where authorization information is obtained for specific authorizations.

- Object authorization

The following are used for authority verification of system objects.

- System pointer authority attribute
- Public authority for the object
- Private authority of current or propagated adopted user profiles in the process
- Private authority for the process user profile

If the process user profile or any of the current or propagated adopted user profiles have the all-object special authorization, any object authorizations required by the operation are implicitly met, and the operation can be performed. When ownership of an object is required for a particular instruction, either the current or propagated adopted user profile or the process user profile can be the owner of the object.

- Special authorizations

If the process user profile or the current or propagated adopted user profiles have the required special authorization, the function can be performed.

- Privileged instruction authorization

If the process user profile or the current or propagated adopted user profile has the required privileged instruction authorization, the instruction can be executed.

- Resource authorization

When a system object is created, the amount of auxiliary storage required for that object is credited to the process user profile. Then, if a user profile exceeds the amount of auxiliary storage allowed, an exception is signaled. The adopted user profiles do not participate in resource authorization.



Program Management

A program is the basic unit of execution in System/38. A program is also the encapsulated and executable form of a program template and logically contains both the function definition and object (data) definitions from the program template. The execution of a program causes a series of functions to be performed against a set of objects. System/38 provides instructions to create, manage, and destroy programs.

PROGRAM CREATION

Programs are created by means of the Create Program instruction. This instruction:

- Accepts the program template as the definition of the program
- Syntax checks the program template according to the syntax rules
- Generates a system object called a program
- Returns diagnostic information through the exception handling mechanism
- Returns a system pointer that addresses the generated program

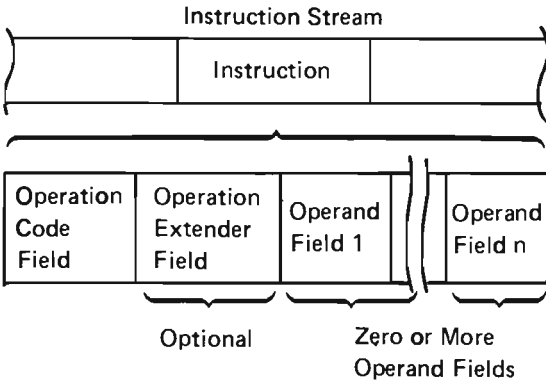
The program template describes the program to be created. The following descriptive items are provided by this operand:

- The template header is fixed at the beginning of the template and contains the creation attributes of the program and the directory to each of the components that make up the program template.
- The template components follow the header and provide the specifications of the program template. Each component is optional and, if present, its location is indicated by the offset values specified in the template header. The components are:
 - Instruction stream
 - Object definition table (ODT)
 - User data

Instruction Stream

The instruction stream defines the set of operations to be performed by the program. It is composed of a series of instructions in an operator-operand format. An instruction can be viewed as a vector of entities, where each unit can be an:

- Instruction operation code (required)
- Operation code extender field (optional)
- Operand of an instruction (zero to four per instruction)

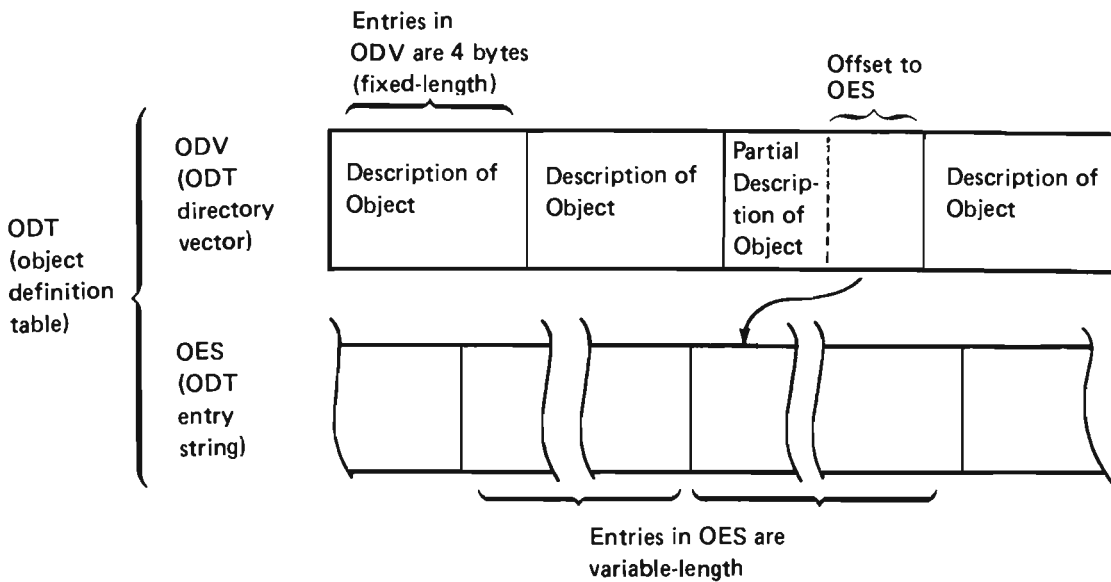


The last 2 bytes of the instruction stream must be an End instruction operation code. This operation code indicates the physical end of an instruction stream and, if executed, functions as a Return External instruction.

Object Definition Table

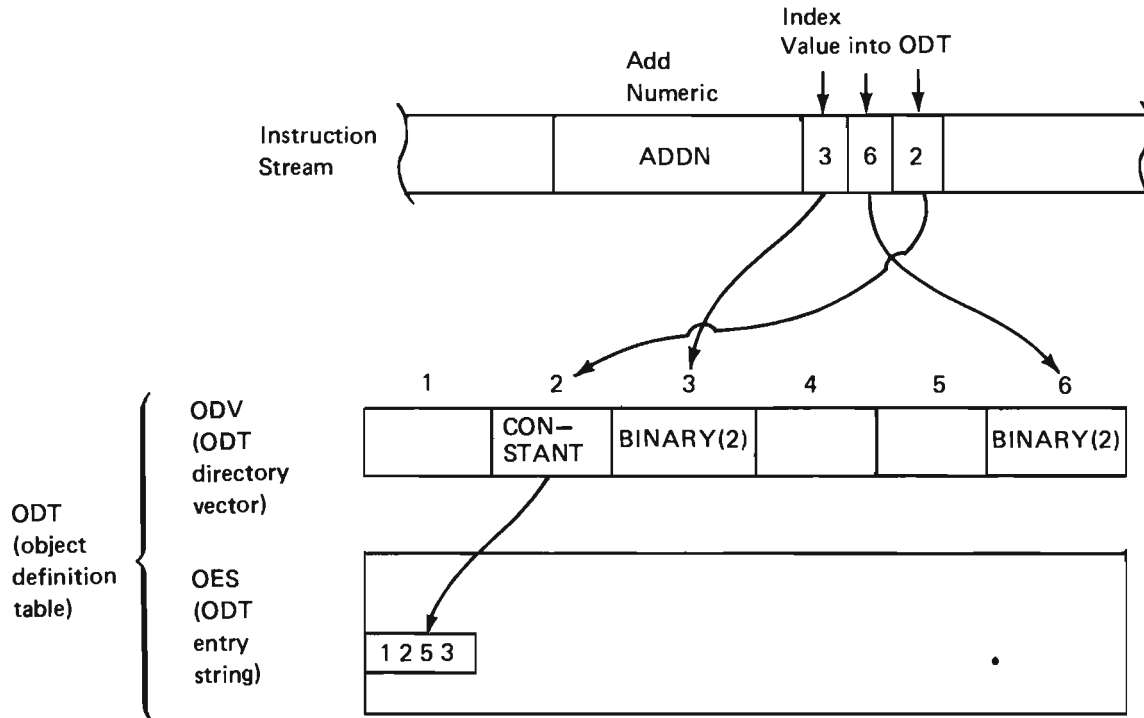
The ODT (object definition table) is the means for defining all objects that are referenced by the operands in the instruction stream. The ODT consists of an ODV (ODT directory vector) and an OES (ODT entry string).

An ODT definition of an object does not actually allocate storage for the object; it does, however, define when and how much storage is to be allocated, as well as define the attributes of the storage.



Each ODV entry defines a program object. Operand fields in instructions (other than implicit references) contain index values to entries in the ODV. These index values are used as program object references; that is, the ODV extent (number of entries) is the address space for an associated instruction stream. An ODV entry can completely specify the attributes for an object, or it can have a partial description of an object and an offset into the OES to an entry that completes the object description. The ODV and OES appear as parts of the program template.

The following illustration shows the relationship between the instruction and the ODT.



User Data

The user must define the format and contents of the user data area. The machine then maintains this area as part of the program. (The user data can be retrieved by materializing the program.)

Program Optimization

The user can request program optimization by specifying the appropriate creation attribute in the program template header. This optional attribute allows for additional processing during program creation, which produces a program that requires less system resource when executed. Program optimization allows a trade-off between the amount of processing required to produce a program and the amount of processing required for its execution.

In general, programs that will be used often should be created with program optimization. The initial cost of creating the program is soon offset by the savings acquired through repeated execution of the program.

PROGRAM DESTRUCTION

The Destroy Program instruction explicitly destroys a program. Program destruction removes all future references to the program from the machine.

PROGRAM MATERIALIZATION

The Materialize Program instruction materializes the components of a specified program template into a target string. The following is a list of the components that can be materialized:

- General program attributes
- Instruction stream
- ODT directory vector (ODV)
- ODT entry string (OES)
- User data
- Object mapping table (OMT)

When a program is materialized, the format of the resulting template is identical to the one used on the Create Program instruction.

Object Mapping Table

The system does default positioning for static and automatic data objects. When a program is created, an OMT (object mapping table) is also created. An entry for each ODT element that maps to a space, whether through a direct or based view, is placed in the OMT.

The OMT consists of a variable-length vector of 6-byte entries. The number of entries in the OMT is identical to the number of ODV entries. Each OMT entry provides a location mapping for the object defined by its associated ODV entry.

PROGRAM OBSERVABILITY DELETION

The Delete Program Observability instruction deletes all of the observable components of a program template, thereby increasing the amount of storage available for programs.

The following components of a program template are deleted:

- Instruction stream
- ODT directory vector (ODV)
- ODT entry string (OES)
- User data
- Object mapping table (OMT)

Computation and Branching

COMPUTATIONAL AND BRANCHING CAPABILITIES

The basic computational and branching capabilities are provided by the following types of instructions:

- Arithmetic instructions compute numeric results.
- Character string-oriented instructions provide special processing for character string operands.
- Boolean instructions perform Boolean (logical) operations on character strings.
- Comparison instructions test the relationship between data items.
- Object movement and conversion instructions copy data from one data item to another.
- Branch instructions change the sequence of program execution either unconditionally or conditionally based on some computational operation.
- Edit instructions change the form of data items for the purpose of displaying the value external to the machine.

COMPUTATIONAL OPERANDS

Computational instructions deal with the numeric and character data residing in spaces. The operands in computational instructions reference data objects, data pointers, and constant data objects. These program objects are defined in the ODT (object definition table). Additionally, an operand can contain a value that is interpreted as an immediate data value.

The operands defined in the ODT specify how addressability is to be established to a set of bytes in a space and how those bytes are to be viewed (attributes) for that operand.

The operands referenced in computational instructions define numeric or character data. Numeric data includes binary, zoned decimal, and packed decimal. Most instructions require operand references to data elements (an element of a data object or an element of an array data object). Some instructions operate on entire arrays; other instruction operands can reference either elements or arrays.

COMPUTATIONAL CHARACTERISTICS

Computational Instructions and Data Descriptions

The data attributes of the instruction operands govern the behavior of the computational instruction set. These data attributes are determined by the definition of the object in the ODT (object definition table). The operation code defines the basic operation to be performed while the operands of the instruction define the exact operational characteristics of the instruction.

For example, if operands A, B, and C are binary scalars, the instruction,

```
ADD NUMERIC A,B,C
```

adds two binary values, operands B and C, and stores the result in binary form in operand A. However, if operands A, B, and C are zoned decimal, the instruction adds two zoned decimal values, B and C, and stores the result in a zoned decimal form in A. There is only one Add Numeric instruction, and it operates on any type of numeric data.

Additionally, the data description provides information concerning the operand length. In the previous example, operands A, B, and C could have been defined as either 2- or 4-byte binary values, or zoned decimal values with 1 to 31 digits. Also, some number of these digits can be considered as fractional digits.

Instructions with operands that can be defined as character or numeric allow any of the numeric data types or the character data type. These instructions (for example, Copy Bytes Right-Adjusted) treat the operand as a string of bytes with a length (in bytes) equal to that defined in the operand's data description. The following examples show how the data length is implied by the data description.

Data Description	Implied Length (bytes)
Character (2)	2
Binary (4)	4
Zoned decimal (5,2)	5
Packed decimal (5,2)	3
Packed decimal (6,2)	4

Generic Computational Operations

Instructions that can operate on data items with any one of several formats are called generic instructions. Many of the computational instructions are generic and thus allow operands of different types, lengths, or precisions.

Computational instructions that allow generic operations have three similar attributes: scalar type, precision, and length.

Scalar Type: Computational instructions with numeric operands, unless otherwise restricted, allow any of the numeric data types (binary, zoned decimal, or packed decimal) on any of the operands with the numeric scalar syntax. For an instruction with multiple numeric operands, the operands need not be of the same type. For example, in the Add Numeric instruction, the receiver can be zoned decimal; the source operands can be binary and packed decimal.

The operation is performed according to the definition of the instruction taking into consideration the formats of the source operands and the format of the result in the receiver.

Precision: Numeric operands, unless otherwise noted, can be defined with any of the precisions allowed for that data type. A binary operand can be 2 bytes or 4 bytes. A decimal operand can specify from 1 to 31 decimal digits. Any number of these decimal digits can be considered as fractional digits.

Computational operations are performed according to the definition of the instruction taking into consideration the precisions of the source operands and yielding a result corresponding to the precision of the receiver or receivers. This means that the number of integer digits and the number of fractional digits participating in the operation (for decimal) are implicitly accounted for in numeric operations.

Length: String operands in computational instructions can be of equal or unequal lengths.

String instructions with operands of unequal length specify one of the following to occur:

- Extra bytes of an operand may be ignored. The length of the shorter operand is chosen as the length of the operation, and extra bytes in the longer operand are ignored (for example, Copy Bytes Left-Adjusted). The operation determines whether the bytes on the left or right end of the operand are ignored.
- Extra (pad) bytes may be supplied to an operand. The length of a longer operand is chosen as the length of the operation, and the shorter operand is padded to this length with some byte value. Each byte in the pad area is given the same byte value. Some instructions allow a user-specified value (Compare Bytes Right-Adjusted with Pad), while other instructions pad with a value specified by the instruction (AND) may be applied to the left or right end of the value depending on the operation being performed.

Additionally, some instructions require an operand to be a minimum length. These instructions use the leftmost bytes to the size required and then ignore all other. This requirement applies to pad operands and indicator operands, which require at least 1 byte and ignore all other bytes in the operand.

Attribute Binding

Attribute binding associates a set of attributes (data view) with the operand of an instruction. Attribute binding typically occurs when a program is created and the attributes of operands are determined from the object definition in the ODT. However, operands in computational instructions can have their data views determined at a time other than at the time a program is created. The attributes of such operands that are not known during program creation time must be supplied to the instruction prior to execution.

The means of supplying a set of attributes to computational operands that refer to data pointers are as follows:

- A data pointer can receive a set of attributes when it is resolved to an external data object. The resolution causes the addressability and attribute set of the data object to be made available through the data pointer.
- A data pointer can be assigned a set of attributes through a Set Data Pointer instruction. The addressability and attributes of the data object addressed by the instruction are assigned to the data pointer.
- A data pointer can be assigned a set of attributes through a Set Data Pointer Attributes instruction. The value of an attribute template specifies the attribute set to be assigned to the data pointer.

The compound substring operand form allows the length attribute to be bound during instruction execution time. The length suboperand can reference a data object whose value may be varied during execution time. The compound operand reference uses the current value as the operand length for a particular execution of the instruction.

Operand Overlap

Operand overlap is a situation in which an operand shares some or all of its space storage with another operand on the same instruction. An operand is nonoverlapping if it does not share any of its space storage with another operand on the same instruction.

Operand overlap can affect the results of an instruction's operation. In situations involving operand overlap, results are valid if they are the same as those obtained when the instruction is executed with nonoverlapping operands; results are invalid if they are different from those obtained when the instruction is executed with nonoverlapping operands. Operand overlap between source operands on an instruction always gives valid results because source operands are not modified by an instruction. But, because receiver operands are modified by an instruction, some cases of operand overlap between receiver and source operands might give invalid results. In cases where it is identified that an operation might give invalid results, it is not guaranteed that invalid results will occur. This is only meant to indicate that the results of such an operation are unpredictable.

Operand overlap can be divided into two subsets, partial overlap and coincident overlap.

Partial Overlap

Partial overlap is a situation in which two operands on an instruction share some, but not all, of their space storage.

Coincident Overlap

Coincident overlap is a situation in which two operands on an instruction share exactly the same space storage.

Operands having coincident overlap can be divided into two subsets.

Identical Operands: This is a situation in which coincident overlap is established through operands that are specified on an instruction with equivalent operand specification fields. Equivalent operand specification fields are the following:

- Equivalent simple operands are those specified with the same ODT reference.
- Equivalent subscripting operands are those specified with the same base ODT reference and an equivalent index. Immediate values specified as an index are equivalent to constants of the same value specified as an index. Indexes specified with the same ODT reference are equivalent.
- Equivalent substrings operands are those specified not only with the same base ODT reference but also with the same equivalent positions and lengths. A position or length specified as an immediate value is equivalent to a corresponding reference to a constant of the same value. Positions or lengths specified with the same ODT references are equivalent.
- Equivalent based operands are those specified with the same base ODT reference and the same basing pointer ODT reference through either implicit or explicit basing. This criterion is combined with that defined for subscripting and substrings in determining equivalency of based subscript or substring operands.

Nonidentical Operands: This is a situation in which coincident overlap is established through operands specified on an instruction with unequivalent operand specification fields. Unequivalent operand specification fields are those that map operands to the same space storage through use of:

- Defined, parameter, or based addressability attributes.
- Variable index, position, or length values on subscripting or substrings operations.
- Constant index, position, or length values specified as separate fields on subscripting or substrings operations.

Overlapping Receiver and Source Operands—Valid and Invalid Results

The following identifies when valid results will occur and when invalid results might occur for cases of operand overlap between receiver and source operands on an instruction.

Overlap Instructions

Overlap instructions, CPYBOLA (Copy Bytes Overlapped Left-Adjusted) and CPYBOLAP (Copy Bytes Overlapped Left-Adjusted With Pad) give valid results for all cases of overlapping operands. They operate as if the source operand was moved to a work area prior to its assignment into the receiver.

Nonoverlap Instructions

Nonoverlap instructions give results for overlapping operands as follows:

Partial Overlapping Operands: All nonoverlap instructions might give invalid results for operands with partial overlap.

Coincident Operands: All standard, (not short form) two operand nonoverlap instructions give valid results for coincident operands.

Nonoverlap instructions with more than two operands for their standard form, including short forms of them, give results for coincident operands as follows:

Identical Coincident Operands: All nonoverlap instructions give valid results for identical coincident operands.

Nonidentical Coincident Operands: The coincident operand overlap program optimization option can ensure that certain nonoverlap instructions will provide valid results even though these instructions contain nonidentical coincident operands. This option, which is specified on the Create Program instruction, causes these nonoverlap instructions to be encapsulated in such a way that the machine will assure valid results. However, additional processor resources are required to assure valid results. Therefore, for this reason, if invalid results can be tolerated when these instructions contain nonidentical coincident operands, the option should not be specified.

Refer to the *Create Program instruction* in the *System/38 Functional Reference Manual* for a list of the instructions affected by the option.

Those MI instructions that have two or more operands and not included in the list, are not affected by the option, and therefore, are encapsulated in a way that they might produce invalid results when the operands are nonidentical coincident.

Examples of Operand Overlap

Assume the following data objects.

DCL	B	BIN(2)		
DCL	BDB	BIN(2)	DEFINED(B)	POSITION(1)
DCL	B2	BIN(2)		
DCL	B3	BIN(4)		
DCL	BDB3	BIN(2)	DEFINED(B3)	POSITION(2)
DCL	BA	BIN(2)	ARRAY(25)	
DCL	BB	BIN(2)	BASED(P)	
DCL	BB2	BIN(2)	BASED(P2)	
DCL	BBA	BIN(2)	BASED(P)	ARRAY(20)
DCL	C	CHAR(200)		
DCL	CDC	CHAR(200)	DEFINED(C)	POSITION(1)
DCL	C2	CHAR(200)		
DCL	C3	CHAR(6)	INTI('ABCDEF')	
DCL	CDC3	CHAR(3)	DEFINED(C3)	POSITION(3)
DCL	CA	CHAR(200)	ARRAY(30)	
DCL	CB	CHAR(200)	BASED(P)	
DCL	CBA	CHAR(200)	BASED(P)	ARRAY(50)
DCL	P	SPP		
DCL	P2	SPP		
DCL	TEN	BIN(2)	CONSTANT(10)	
DCL	MYTEN	BIN(2)	CONSTANT(10)	

Overlapping operands on overlap instructions: Valid results will be obtained from the following operation.

CPYBOLA CDC3,C3 (Operands 1 and 2 partially overlap)

This gives a result of CDC3 = 'ABC'.

Partially overlapping operands: Invalid results might be obtained from the following operations.

CPYBLA CDC3,C3
 ADDN B3,BDB3,B (Operands 1 and 2)
 OR CDC3,C,C3 (Operands 1 and 3)

Coincident operands on two operand instructions: Valid results will be obtained from the following operations.

CPYBLA C,C
 SETSPP P,C
 CPYBLA C,CB

Identical coincident operands on three operand instructions: Valid results will be obtained from the following operations.

Identical simple operand references:

ADDN B,B2,B (Operands 1 and 3)
 OR C,C,CA(1) (Operands 1 and 2)

Identical subscripting operand references:

ADDN BA(B),B,BA(B) (Operands 1 and 3)
 ADDN BA(10),BA(TEN),B (Operands 1 and 2)
 ADDN BA(TEN),BA(TEN),B (Operands 1 and 2)
 OR CA(1),CA(1),C (Operands 1 and 2)

Identical substringing operand references:

OR C(B,BB),CB,C(B,BB) (Operands 1 and 3)
 OR C(TEN,1),C(10,1),C2 (Operands 1 and 2)
 OR C(1,10),CA(1),C(1,10) (Operands 1 and 3)

Identical based operand references:

ADDN BB,BB,B (Operands 1 and 2)
 ADDN P->BBA(TEN),B,P->BBA(10) (Operands 1 and 3)
 OR P->CB,C,CB (Operands 1 and 3)

OR CB(TEN,1),CB(10,1),C (Operands 1 and 2)
 OR CBA(TEN),CB,CBA(10) (Operands 1 and 3)
 OR CBA(B),CB,P->CBA(B) (Operands 1 and 3)

OR P->CB(1,10),C,P->CB(1,10) (Operands 1 and 3)

Nonidentical coincident operands on three operand instructions: Invalid results might be obtained from the following operations.

Nonidentical simple operand references:

ADDN B,BDB,BA(1) (Operands 1 and 2)
 OR C,CA(1),CDC (Operands 1 and 3)

Nonidentical subscripting operand references:

CPYNV B,1
 ADDN BA(1),B2,BA(B) (Operands 1 and 3)
 CPYNV B,10
 CPYNV B2,10

ADDN BA(B),BA(5),BA(B2) (Operands 1 and 3)

OR CA(TEN),CA(MYTEN),C (Operands 1 and 2)

Nonidentical substringing operand references:

CPYNV B,1
 OR C(1,5),C(B,5),CA(5) (Operands 1 and 2)
 CPYNV B,10
 CPYNV B2,10

OR C(B,5),C2,C(B2,5) (Operands 1 and 3)

OR C(TEN,1),C(MYTEN,1),C2 (Operands 1 and 2)

Nonidentical based operand references:

SETSPP P,B
 ADDN BB,B,BA(1) (Operands 1 and 2)
 ADDN B,BBA(1),B2 (Operands 1 and 2)
 SETSPP P,B
 SETSPP P2,B

ADDN BB,P2->BB,BA(1) (Operands 1 and 2)
 ADDN BB,BB2,BA(1) (Operands 1 and 2)

ADDN P2->BB,B,P2->BB2 (Operands 1 and 3)

Avoiding Invalid Results

The invalid results that might occur in the execution of instructions that do not allow for operand overlap can be avoided if the source operands are moved to a temporary area.

Where

ADDN B3,BDB3,BA(1)

might give invalid results due to partial overlap between operands 1 and 2,

CPYNV B2,BDB3
 ADDN B3,B2,BA(1)

will give valid results.

The following is another way to avoid invalid results.

For those instructions that do not allow for nonidentical coincident operand overlap, invalid results can be avoided through use of the coincident operand overlap program optimization option. This option, when specified on the Create Program instruction, causes certain instructions to be encapsulated in a way that ensures that they produce valid results should nonidentical coincident operand overlap occur.

Refer to the *Create Program instruction* in the *System/38 Functional Reference Manual—Volume 1* for a list of the affected instructions.

Optional Computational Instruction Forms

Many System/38 computational instructions have optional forms that allow additional functions of the base instruction. Each optional form has common syntax and common operational rules independent of the base instruction function. The fact that the instruction is an optional form is specified in the operation code.

The functions of the optional forms are independent of one another. This allows a single instruction to make use of more than one form.

Short Form

The short form can be used with some of the arithmetic, Boolean, and character string instructions that perform an operation on source operands and place the result in a receiver operand. The short form does not alter the operation of the base instruction.

The short form is indicated in the operation code of the instruction by including the character S with the mnemonic of the base instruction.

The short form of an instruction is used when one of the source operands is specified as the receiver operand and thus need not be specified twice. The standard form of the instruction, in contrast, requires a separate specification of the source operand and the receiver operand. For example:

- To perform the addition operation: $A = A + B$

ADDN A,A,B (Standard form)

or

ADDNS A,B (Short form)

- To perform the AND operation: $X = X \& Y$

AND X,X,Y (Standard form)

or

ANDS X,Y (Short form)

Round Form

Some arithmetic instructions that operate on decimal operands allow optional round forms. When the round form is specified, the result of the operation is rounded before being placed in the receiver operand.

The standard form of the arithmetic instruction, after decimal point alignment, truncates any fractional digits as required in the receiver operand. The round form alters the operation of the base instruction by rounding the value of the result if the most significant digit of the truncated fraction is greater than or equal to 5. The rounding operation involves adding a value of 1 to the least significant digit of the truncated result. The carry (if any) is propagated to the left in the result. If the result is negative, the rounding operation involves a subtraction rather than an addition. Rounding of the result occurs before any size exception checks are made.

The round form is indicated in the operation code of the instruction by adding the character R to the end of the mnemonic of the base instruction. For example, assume three numeric elements with the following values:

A	ZONED(5,2)	'123.45'
B	PACKED(4,3)	'1.324'
C	PACKED(5,4)	'4.2312'

Executing the Add Numeric instruction produces the following result:

ADDN A,B,C (Standard form)

would be $A = 005.55$

Executing the Add Numeric (Round) instruction produces the following result:

ADDNR A,B,C (Round form)

result is $A = 005.56$

Branch Form

Some System/38 computational instructions have branch forms that allow multiway conditional branching based on the status of the instruction execution. This form is in addition to the standard form of the instruction that does not include branching. For example:

ADDN	A,B,C	(Standard form—add and leave no status about result)
or		
ADDNB (P)	A,B,C,Z	(Branch form—add and branch to Z if result is positive)

The fact that an instruction is a branch form is indicated in its operation code by adding the character B to the mnemonic of the base instruction. The extender field and one or more branch target operands are used in a common fashion for all optional branch form instructions to define the branch options and the branch locations, respectively.

For more details on the branch forms of instructions, refer to *Conditional Branching* later in this chapter.

Indicator Form

In addition to branching based on the result of an instruction, the user can set one or more switches that can be tested later.

These switches (called indicators) can be set based on the results of an instruction. Some instructions allow an indicator form in which one or more 1-byte character strings can be set. For example:

ADDN	A,B,C	(Standard form—add and leave no status about the result)
or		
ADDNI (P)	A,B,C,D	(Indicator form—add and assign a value of hex F1 to D if the result is positive or a value of hex F0 if the result is not positive)

The indicator form is an option in the instruction operation code and is indicated by adding the character I to the mnemonic of the base instruction. The extender field defines the number of indicator operands to be used and the conditions to be tested in order to assign values to the indicators.

If the resulting status of the instruction matches the condition specified in the extender field, the indicator operand is given a value of hex F1. If the status does not match the tested condition, the indicator operand is given the value of hex F0.

The indicator form and the branch form are mutually exclusive.

Some instructions require either the branch or the indicator form (for example, the Compare instructions). These instructions require the operation code extender field and at least one branch or indicator target.

ARITHMETIC OPERATIONS

System/38 arithmetic instructions are primarily designed to compute numeric results; they operate on numeric scalars of the following types: binary, zoned decimal, and packed decimal.

The result of an arithmetic operation is placed in the receiver based on the characteristics of the result and the attributes of the receiver.

Binary Computation

The following rules apply to binary operands in arithmetic instructions:

- An attempt to complement the maximum negative value causes a size exception.
- Truncation is performed on the left; a size exception is signaled when significant high order digits are lost. Significant high-order digits are lost if all of the bits truncated on the left are not equal to the sign bit of the truncated result. The rightmost 16 or 32 bits of the result are placed in the receiving field for 2-byte and 4-byte binary receivers, respectively.
- Padding is done on the left by propagating the sign from the high-order bit.
- A zero result in a computation has a positive sign.

Packed Decimal Computation

The following rules apply to packed decimal operands in arithmetic instructions.

- All digits are checked for valid encoding of hex 0 through hex 9. If an invalid digit is detected, a decimal data exception is signaled.
- All signs are checked for valid encoding as follows:
 - Hex B or hex D means the value is negative.
 - Hex A, hex C, hex E, or hex F means the value is positive.

An invalid sign causes the decimal data exception to be signaled.

- If alignment is necessary, source operands are aligned based on the assumed decimal point by truncating digits or padding with zeros on the right. Fractional digits that can affect the value to be placed in the receiving field participate in the calculation of the result.
- If necessary, the operands are expanded to the length needed to perform the operation by padding with zeros on the left.
- When aligning a source operand, if more than 31 decimal digits are required to contain the aligned value, a decimal point alignment exception is signaled. The exception is signaled when nonzero digits must be truncated from the left end of the aligned value to conform to a 31-digit field.
- Length adjustment and decimal point alignment are performed at the left and right ends of the result, respectively, by truncating digits or padding with zeros to match the precision of the receiver operand. If nonzero digits are lost in truncating at the left, a size exception is signaled. If the optional round form of an instruction is being used, rounding on the right end occurs if any digits are truncated.
- The sign of a receiver operand value is always set independently of any truncation and/or padding that could have taken place (that is, in the rightmost 4 bits of the rightmost byte of the result).
- Arithmetic results are given the preferred sign (hex F for positive and hex D for negative). Zero values are given the preferred positive sign.
- The four high-order bits of the leftmost byte of a packed receiver field contains a value of hex 0 when the field contains an even number of digits.

Zoned Decimal Computation

The rules for zoned decimal operands in arithmetic instructions are the same as those for packed decimal operands. In addition, the zone portion of each nonsigned digit in the receiver operand is set to a hex F.

Floating-Point Computation

The following rules apply to floating-point operands in computational instructions.

- Floating-point operations are performed for instructions for which any of the operands are specified as floating-point. Fixed-operations are performed for instructions for which all operands are specified as either fixed-point binary or fixed-point decimal.
- Certain computational attributes for floating-point operations can be controlled on a process basis through use of the Store and Set Computational Attributes instruction. A default set of computational attributes is in effect when a process is initiated. The computational attributes can be set by an invocation and are in effect for subsequent invocations unless changed with the Store and Set Computational Attributes instruction. When processing returns to an invocation from subsequent invocations, the computational attributes are reset to the attributes that were in effect when the invocation gave up control. Refer to the *Store and Set Computational Attributes* instruction in the *System/38 Functional Reference Manual* for details about managing the computational attributes for a process.
- Alignment of the binary point, if necessary, is performed according to the requirements of the particular operation. Refer to the *System/38 Functional Reference Manual* for the algorithm used by specific instructions.

- The operands are expanded to the length needed, or converted to the type needed according to the requirements of the particular operation. This occurs when an intermediate result is formed.

When all operands are floating-point and of the same length, operations are performed as if to infinite precision. This occurs unless specified otherwise in the particular instruction. These operations are only subject to one rounding error when the result is stored in the receiver.

When at least one of the operands is floating-point, but all operands are not floating-point of the same length, operations may not be performed as if to infinite precision. The result is formed using the short or long format, depending upon the precision required, to adequately provide for the requirements of the specified operation. Conversions of input values to the floating-point format appropriate for the operation are subject to rounding errors when the input value is not an integer value. This can only occur for decimal fields with fractional digit positions. The calculation of the result is also subject to a rounding error. A rounding error can also occur when the result is stored in the receiver. Therefore, these operations are subject to multiple rounding errors in the value stored in the receiver.

- Floating-point operations produce an intermediate result that is a normalized number, signed zero, infinity, or an NaN floating-point value.

When the result is a normalized number, it is produced as if it were infinitely precise and unlimited in exponent range, unless stated otherwise in the specific instruction, or the operation involved conversions as previously stated. The normalized number may be the result of internal calculations that produced an internal result that did not satisfy the definition of a normalized number. In this case, a normalization operation is performed on the internal result; this operation appropriately shifts the bits of precision, while adjusting the exponent, until the leading one bit is just to the left of the binary point. The exponent is regarded as if its range were unlimited.

For an intermediate result value of signed 0, infinity, or an NaN, assigning this result to the receiver simply means representing its value in the receiver format.

If the receiver is fixed-point, an infinity or an NaN value causes the invalid floating-point conversion exception to be signaled. See the discussion of floating-point exceptions, which follows for details. A signed 0 value, is represented as the appropriate 0 value in the receiver format.

If the receiver is floating-point, the assignment of the result does not alter the result value to another type of floating-point value, as can happen for an intermediate result that is a normalized number.

When an intermediate result value of a normalized number is assigned to the receiver, the result may require an adjustment because it is outside the range of normalized numbers that can be represented in the receiver.

If the receiver is fixed-point, the normalized number is converted to the format of the receiver. Also, it is adjusted to the precision of the receiver under control of the rounding mode currently in effect for the process unless overridden by specifying the optional round form of an instruction. The optional round form of an instruction is only allowed for operations that specify fixed-point receivers. Due to the possible adjustment in precision, the floating-point inexact result exception condition can be detected. Additionally, the assignment of the result value to the receiver can result in the signaling of the invalid floating-point conversion exception.

If the receiver is floating-point, the system performs several steps to provide for properly representing the normalized number in the receiver.

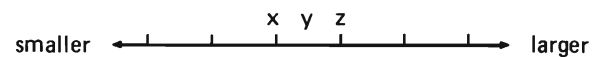
The initial step is to check for the floating-point underflow condition. This is done by verifying that the signed exponent of the result is not less than the minimum value (-126 for short format or -1022 for long format) for representation of normalized numbers in the receiver format. If it is not less than the minimum value, the operation continues with the rounding step. If it is less than the minimum value, a floating-point underflow exception condition may or may not be detected depending upon the mask state of the exception. When the exception is masked, the intermediate result is adjusted, as if to infinite precision, to a denormalized number appropriate for the format of the receiver, and the operation continues with the rounding step. The intermediate denormalized result is produced by shifting the significand of the intermediate result right and incrementing the exponent until the exponent attains the receiver format's fixed value for denormalized numbers (-126 for short format or -1022 for long format). As a result of the rounding step, the floating-point underflow occurrence indicator is set if the intermediate denormalized result cannot be represented in the receiver format. In this case, the intermediate denormalized result may be adjusted back to a normalized number, to signed 0, or remain a denormalized number. In any case, the result is no longer exact and, therefore, forces the floating-point underflow occurrence indicator to be set.

The next step, rounding, chooses a representation in the format of the result field for the intermediate result. The intermediate result is regarded to be of infinite precision. The rounding mode currently in effect controls the adjustment of the result value. If the adjustment of the result value causes a loss of nonzero digits from the significand, a floating-point inexact result exception condition is detected. As previously noted, detection of the inexact result condition on the adjustment of an intermediate denormalized result forces the setting of the floating-point underflow occurrence indicator regardless of the value to which the result is adjusted. In conjunction with the process of rounding, a check for the floating-point overflow condition is performed. This is done by verifying that the signed exponent of what is the rounded result, or what would have been the rounded result if the exponent range was unlimited, is not greater than the maximum value (127 for short format or 1023 for long format) for representation of normalized numbers in the receiver format. If it is not, the operation continues with the final step, which assigns the value of the intermediate result into the receiver. If it is, the floating-point overflow exception condition is detected. See the discussion of floating-point exceptions provided below for details.

The final step is to represent the value of the adjusted intermediate result in the floating-point element specified as the receiver. The adjusted value of the intermediate result may still be a normalized number, or it may have been altered to a denormalized number or signed 0.

- Floating-point fields can only represent numeric values as normalized numbers, denormalized numbers, or signed 0. Therefore, the concept of an unnormalized number (one which would allow for a variable exponent in conjunction with one or more leading 0 bits prior to the first significand 1 bit) does not exist and cannot be represented.

- Four floating-point rounding modes are supported. For example, assume y is the infinitely precise number that is to be rounded. In addition, assume that y is bracketed most closely by x and z , where x is the largest representable value less than y , and z is the smallest representable value greater than y . Note that x or z may be infinity. The following diagram shows this relationship of x , y , and z on a scale of numerically progressing values where the vertical bars denote values representable in a floating-point format.



If y is not exactly representable in the receiving field format, the rounding modes change y as follows:

- Round to nearest with round to even in case of a tie is the default rounding mode in effect when a process is initiated. For this rounding mode, y is rounded to the closer of x or z . If they are equally close, the even one (the one whose least significant bit is a 0) is chosen. For the purposes of this mode of rounding, infinity is treated as if it were even. Except when y is rounded to a value of infinity, the rounded result will differ from the infinitely precise result by at most half of the least significant digit position of the chosen value. This rounding mode differs slightly from the decimal round algorithm performed for the optional round form of an instruction. This rounding mode would round a value of 0.5 to 0, whereas the decimal round algorithm would round that value to 1.
- Round toward positive infinity mode indicates that directed rounding upward is to occur. For this mode, y is rounded to z .
- Round toward negative infinity mode indicates that directed rounding downward is to occur. For this mode, y is rounded to x .
- Round toward zero mode indicates that truncation is to occur. For this mode, y is rounded to the smaller (in magnitude) of x or z .

- Conversions between floating-point integers and fixed-point integer formats (binary or decimal with no fractional digits) will be exact, unless the number of significant digits of a source decimal value exceeds the precision constraints of a floating-point receiver.
 - Conversions between floating-point numbers and fixed-point decimal numbers are performed such that all the decimal digits specified for the decimal number are either used in or produced from the conversion. However, the precision provided by floating-point fields is not as great as that provided by decimal fields. The short format provides unique representation of a maximum of 7 significant decimal digits of precision, and the long format provides for a maximum of 15. The leftmost nonzero digit of the decimal number is considered the start of the significant digits of the number.
 - When the system converts a fixed-point decimal value to floating-point, significant digits of the source decimal field beyond 7 (for short format) or 15 (for long format) may not be saved in the floating-point field; their only function is to provide for rounding and uniqueness of the conversion.
 - When the system converts a floating-point value to fixed-point decimal, significant digits produced in the receiver beyond the first 7 (for short format) or the first 15 (for long format) are correct relative to the specific source floating-point value. These digits, which exceed the precision constraints of the floating-point field, serve to provide for uniqueness of conversion and should be considered only as precise as the calculations that produced the floating-point number. The floating-point inexact result exception provides a means of detecting loss of precision in floating-point calculations.
 - When a round to nearest operation occurs, conversion from floating-point to decimal and back to floating-point is identical as long as the decimal string provides for a precision of 9 significant decimal digits for short format conversions and 17 significant decimal digits for long format conversions.
 - The sign of a product or a quotient is the exclusive OR of the operands' signs. The sign of a sum or of a difference differs from at most one of the operands' signs following the standard rules of algebra. The previous rules apply even when operands or results are 0 or infinite. The only exception is when the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly 0; the sign of that sum (or difference) depends on the current rounding mode for the process.
 - For round toward negative infinity mode, the sign is -.
 - For all other rounding modes, the sign is +.
- Conversion operations, including those between floating-point and fixed-point, preserve the sign of 0 if the result of the conversion operation can be represented in the receiver. This cannot be done for conversion of a negative zero value into a fixed-point binary field, as that data type has no representation for negative zero.
- Masked NaNs in source operands are moved into floating-point receivers. Unmasked NaNs in source operands are changed to masked NaNs and moved into floating-point receivers when the floating-point invalid operand exception is masked. If more than one source operand is a NaN, then the NaN moved into the receiver is the NaN with the largest fraction field value. For the purpose of the comparison, all of the input NaNs are considered masked. Additionally, if the floating-point receiver is longer than the source field that supplied the NaN, the resulting masked NaN is set with the fraction field value from the source padded with 0 bits on the right out to the float receiver fraction field length. The sign field of the NaN set into the receiver is preserved with the value it contained in the source.

- Unmasked NaNs in source operands force detection of the floating-point invalid operand exception. An exception to this is when a numeric value operation copies the value represented in a source floating-point element to a receiver of the same format. This is defined as a simple move operation and the invalid floating-point operation exception is not detected if the source represents an unmasked NaN.
- Infinity values in source operands can be used in arithmetic operations according to the standard rules of algebra. They produce a correctly signed infinity value in the receiver, unless otherwise specified by a specific instruction. Negative infinity compares less than every finite value, and every finite value compares less than positive infinity.

The following are examples of floating-point computations. See the discussion of floating-point elements for an explanation of the syntax used in these examples.

- This example shows an add operation ($A = B + C$) executed as an Add Numeric instruction (ADDN A,B,C) involving all short format operands.

Initially, the add operation is shown as:

	Element Value In Hexadecimal	Conceptual Numeric Value	Comments
C	3F800000	0 NUM +1.000000000000000000000000	Value of C, +1
B	40400000	+1 NUM +1.100000000000000000000000	Value of B, +3
<hr/>			
A	40800000	+2 NUM +1.000000000000000000000000	(add operation produces) Result value +4

Internally, the addition operation is shown as:

C	+1. NUM + 0.100000000000000000000000	Aligned value of C
B	+1. NUM + 1.100000000000000000000000	Value of B
<hr/>		
	+1 NUM +10.000000000000000000000000 .	(add operation produces) The internal result (normalization operation produces)
	+2 NUM +1.000000000000000000000000 .	The intermediate result (rounding operation produces)
A	+2 NUM +1.000000000000000000000000	Value of A

- This example shows an add operation (A = B + C) executed as an Add Numeric instruction (ADDN A,B,C) involving all short format operands with the round to nearest rounding mode in effect.

Initially, the addition operation is shown as:

	Element Value In Hexadecimal	Conceptual Numeric Value	Comments
C	3FFFFFFE	0 NUM +1.11111111111111111110	Value of C, almost +2
B	407FFFFC	+1 NUM +1.11111111111111111100	Value of B, almost +4 (add operation produces)
<hr/>			
A	40BFFFFE	+2 NUM +1.01111111111111111110	Result value, almost +6

Internally, the addition operation is shown as:

C	+1 NUM + 0.11111111111111111110	Aligned value of C
B	+1 NUM + 1.11111111111111111100	Value of B (add operation produces)
<hr/>		
	+1 NUM +10.1111111111111111110110	The internal result (normalization operation produces)
	•	
	+2 NUM +1.0111111111111111110110	The intermediate result (rounding operation produces)
	•	
A	+2 NUM +1.01111111111111111110	Value of A

The following floating-point exception conditions can be detected during floating-point operations:

- Floating-point overflow
- Floating-point underflow
- Floating-point zero divide
- Floating-point inexact result
- Floating-point invalid operand

Associated with each of these exceptions is a set of mask and occurrence bits.

The mask bit determines whether an exception is signaled. If the mask bit is 0, the exception is considered to be masked and is not signaled. If the mask bit is 1, the exception is considered to be unmasked and is signaled. When a process is initiated, the default mask bit values specify that the floating-point inexact result is masked, and all other exceptions are unmasked. The mask bits can be tested and set with the Store And Set Computational Attributes instruction. The result of float exceptions can vary depending upon whether the exception is masked or unmasked.

The occurrence bit records the occurrence of the exception condition whether or not the exception is masked when it is detected. A value of 1 is set to indicate an exception condition has occurred. A value of 0 indicates that the exception condition has not occurred. When a process is initiated, the default occurrence bit values are all 0's. The occurrence bits can be set (0 or 1) with the Store And Set Computational Attributes instruction.

Conversion operations from binary floating-point to other than binary floating-point format can cause the invalid floating-point conversion exception to be signaled. This exception cannot be masked and has no associated occurrence bit. For details on this exception, refer to the *System/38 Functional Reference Manual*.

Floating-Point Overflow

A floating-point overflow condition is detected whenever the largest finite number that can be represented in the format of the floating-point receiver is exceeded in magnitude by what would have been the rounded floating-point result if the range of the exponent was unlimited. For this to occur, the signed exponent of the result must exceed 127 for a short format receiver or 1023 for a long format receiver.

The occurrence of the floating-point overflow condition is indicated through the setting of the floating-point overflow occurrence bit.

The setting of the floating-point overflow mask affects the result of the operation as follows:

- If the exception is masked, the exception is not signaled, the floating-point inexact result is detected, and the result of the operation is determined by the rounding mode and the sign of the intermediate result as follows:
 - Round to nearest mode produces infinity with the sign of the intermediate result.
 - Round toward zero mode produces the receiver format's largest finite number with the sign of the intermediate result.
 - Round toward negative infinity mode produces the receiver format's largest finite number for positive overflows, and negative infinity for negative overflows.
 - Round toward positive infinity mode produces the receiver format's most negative finite number for negative overflows, and positive infinity for positive overflows.

If the exception is not masked, the exception is signaled, the value of the receiver operand is unpredictable, and the exception data available depends upon the operation being performed.

- An overflow detected on a conversion operation from the long to the short floating-point format results in a long format value rounded to a short format precision to be provided in the exception data.
- An overflow detected on a conversion operation from a decimal form of a floating-point value, on the scaling operation performed in the Scale instruction, or on certain cases of the Compute Math Function instruction causes a long format system default masked NaN value to be provided in the exception data.
- An overflow detected on an arithmetic operation causes a long format value to be provided in the exception data. For a short format receiver, the long format value provided is rounded to short format precision. For a long format receiver, the long format value provided is a correctly rounded significand, a correct sign, and a modified exponent. The modified exponent is set from the overflowed normal biased exponent minus a bias adjust value of 1536. This bias adjust value (1536) translates overflowed biased exponents as nearly as possible to the middle of the representable biased exponent range for the long format. An exception handler can then be provided with appropriate information for later reconstruction of the correct result. The following diagram summarizes the relationships among the overflowed values for the signed exponent, the normal biased exponent, and the modified biased exponent.

	Overflowed Exponent		
	Signed	Normal Biased	Modified Biased
Minimum Value	1024	2047	511
Maximum Value	2047	3070	1534

Floating-Point Underflow

A floating-point underflow condition may be detected when a result that is not 0 is examined prior to rounding and is found to have too small an exponent to be represented in the format of the receiver without being denormalized. For the underflow condition to exist, the signed exponent of the result must be less than -126 for a short format receiver or less than -1022 for a long format receiver.

The value (0 or 1) of the floating-point underflow mask bit affects the detection of the exception condition as well as the result of the operation.

- If the exception is masked (bit value equals 0), the underflow condition is only detected and indicated through the setting of its related occurrence bit if the denormalized number for the intermediate result cannot be exactly represented in the floating-point receiver. In this case, the floating-point receiver is set with a value that is produced by first denormalizing the unrounded result, then rounding, then moving the result to its receiver. Only the occurrence bit for underflow is set, the underflow exception is not signaled.
- If the exception is not masked (bit value equals 1), the floating-point underflow condition is indicated through the setting of the floating-point underflow occurrence bit and the exception is signaled whenever the signed exponent of the result is too small for a normalized number to be represented in the receiver. The value of the receiver operand is unpredictable, and the exception data available depends upon the operation being performed.
 - A long format value rounded to short format precision is available if an underflow condition is detected on a conversion operation from the long to the short floating-point format.
 - A long floating-point system default masked NaN value is available if an underflow condition is detected on a conversion from a decimal form of a floating-point value, on the scale operation performed for the Scale instruction, or on the Compute Math Function instructions.

- A long format value is available if an underflow condition is detected on an arithmetic operation. For a short format receiver, the long format value available is rounded to short format precision. For a long format receiver, the long format value available is a correctly rounded significand, a correct sign, and a modified exponent. The modified exponent is set from the underflowed normal biased exponent plus 1536. This bias adjust value translates underflowed biased exponents as nearly as possible to the middle of the representable biased exponent range for the long format. This provides the appropriate information to an exception handler for later reconstruction of the correct result. The following diagram summarizes the relationship between the underflowed values for the signed exponent, the normal biased exponent, and the modified biased exponent.

	Underflowed Exponent		
	Signed	Normal Biased	Modified Biased
Maximum Value	-1022	1	1537
Minimum Value	-2148	-1125	411

The maximum underflowed exponent value in the previous diagram occurs when rounding of the underflowed value increases its value back above the underflow threshold.

The minimum underflowed exponent value in the previous diagram occurs when two minimum valued denormalized numbers are multiplied together to produce an intermediate result with a signed exponent of the indicated value.

Floating-Point Zero Divide

A floating-point zero divide condition is detected for floating-point division if the divisor is zero and the dividend is a finite nonzero number. The floating-point zero divide condition is indicated through the setting of the floating-point zero divide occurrence bit. The setting of the floating-point zero divide mask bit affects the result of the operation.

- If the exception is masked (bit value equals 0), the result of the operation is a correctly signed infinity value (exclusive OR of the operands' signs), and the exception is not signaled.
- If the exception is not masked (bit value equals 1), the operation is suppressed, and the exception is signaled.

Floating-Point Inexact Result

A floating-point inexact result condition is detected (in the absence of the floating-point invalid operand exception condition) if the rounded result of an operation is not exact.

- The rounded result of an operation is not exact when the rounding operation on an intermediate result causes a loss of nonzero significand digits in representing the value of the result in the receiver. This applies to fixed-point receivers of floating-point operations as well as to floating-point receivers.
- The result of an operation is not exact when a floating-point overflow condition occurs while that condition is masked. The receiver is set at either infinity, or the receiver format's largest magnitude finite number.

The floating-point inexact result condition is indicated by the floating-point inexact result occurrence bit.

If the floating-point inexact result exception is either masked or unmasked, the rounded or overflowed result is moved to the receiver. If the exception is masked (bit value equals 0), the exception is not signaled. If the exception is not masked (bit value equals 1), the exception is signaled.

Floating-Point Invalid Operand

A floating-point invalid operand condition is detected when an operand is invalid for the operation to be performed:

- A source operand is an unmasked NaN.
- Addition of infinities of different signs or subtraction of infinities of the same sign.
- Multiplication of 0 times infinity.
- Division of 0 by 0, or infinity by infinity.
- Computing a math function for certain operand combinations. Refer to the *System/38 Functional Reference Manual* for details concerning the Compute Math Function instructions.
- Floating-point values compare unordered, and no branch or indicator options are specified for the unordered, negation of unordered, equal, or negation of equal conditions when the Compare Numeric Value instruction is executed.
- An unordered resultant condition occurs on a computational instruction when the result is a NaN, and branch or indicator conditions are specified, but none of the unordered, negation of unordered, zero, or negation of zero conditions are selected.

The floating-point invalid operand condition is indicated by the floating-point invalid operand occurrence bit.

The value (0 or 1) of the floating-point invalid operand mask bit affects the result of the operation.

- If the exception is masked (bit value equals 0), the exception is not signaled.

If the exception condition is detected on a comparison operation, and the condition is caused by an invalid operand associated with the specified branch or indicator options, the receiving field (if applicable) is left intact with the calculated result of the operation.

If the exception is detected during an operation in which a floating-point result is to be stored, the result of the operation is a masked NaN value.

- If the exception was due to one or more operands being an unmasked NaN, then the input NaN with the largest fraction field value is propagated into the receiver with its mask state set to masked. All of the input NaNs are considered masked for the compare operation. Additionally, if the receiver format is longer than the source field that supplied the NaN, the resulting masked NaN is set with the fraction field value from the source, and padded with 0 bits on the right out to the float receiver fraction field length.
 - If the exception was not due to an operand being an unmasked NaN, then the resulting masked NaN that is propagated into the receiver is the system default NaN which is appropriately represented in the receiver format.
- If the exception is not masked, the exception is signaled and the value of the receiver operand is unpredictable. The exception data available indicates whether or not the exception was detected due to an invalid branch or indicator option.

Arithmetic Instructions

The following table shows the function of each arithmetic instruction; in each case, the result is placed in the receiver operand. For a detailed instruction description, refer to the *System/38 Functional Reference Manual*.

Instruction	Function
Add Numeric	Forms the algebraic sum of two numeric values.
Subtract Numeric	Forms the algebraic difference of two numeric values.
Multiply	Forms the algebraic product of two numeric values.
Divide	Forms the algebraic quotient of two numeric values.
Divide with Remainder	Forms the algebraic quotient and the remainder of two numeric values.
Remainder	Forms the algebraic remainder as the result of dividing two numeric values.
Extract Magnitude	Forms the value of a numeric operand as a positive quantity.
Negate	Changes the sign of a numeric operand (positive values become negative and negative values become positive).
Scale	Multiplies a numeric operand by $B^{**}n$. The quantity B is 10 for decimal, and 2 for binary. The quantity n is the scale factor.
Compute Math Function Using One Input Value	Computes a floating-point result by applying the mathematical function requested in the controls operand to the specified source operand.
Compute Math Function Using Two Input Values	Computes a floating-point result by applying the mathematical function requested in the controls operand to the specified source operands.

CHARACTER STRING OPERATIONS

The character string instructions operate on strings of characters. String operations are performed by considering corresponding bytes of the source operand(s) and the receiver operand.

Source string operands that are shorter than the receiver length are padded on the right end with a pad value of hex 40 (blank).

Source string operands longer than the receiver are truncated on the right to the length of the receiver.

Operands in string instructions can include a compound operand that defines a substring of a character string. This operand includes the string, an index value, and a length value. These compound substring operands can be used as source operands or receiver operands.

Compound substring operands optionally allow or disallow references to a substring with a length value of zero (null substring). Null substring references are supported on only a subset of the instructions that support character data as operands. To determine if a particular instruction provides this support, refer to the *System/38 Functional Reference Manual*.

Character String Instructions

The following instructions operate on character strings.

Concatenate: This instruction forms a character string by joining the second operand string to the right of the first operand and placing the result in the receiver operand.

Translate: This instruction transforms strings of characters from one encoding to another.

Four character string operands are specified:

- Receiver string
- Source string

- Position string
- Replacement string

Characters of the source string are compared with a position string for equality. If a match is found, the character in the corresponding position in the replacement string is copied to the receiver string; otherwise, the source character is moved to the receiver.

Scan: This instruction searches a character string for the first occurrence of a specified substring. The binary element receiver is set to a binary value indicating the relative location of the leftmost character of the matching string. If no match is found, a zero value occurs. If the locations for multiple occurrences of the substring are desired, the receiver must be a binary array.

Scan with Control: This instruction searches a character string for a character code that has a specified relation (high, equal, or low) to a specified character code value. This instruction can scan for 1- or 2-byte character codes. The user can specify that the string that will be scanned should contain both 1- and 2-byte character codes with special mode control characters indicating which mode (1- or 2-byte) should be used to interpret the string. The user can also specify that the string that will be scanned should contain only one type of character code, all 1-byte or all 2-byte, with no imbedded mode control characters. An optional escape can be requested on 1-byte values less than hex 40.

Extended Character Scan: This instruction searches a character string for a specified character code. This instruction can scan for 1- or 2-byte character codes. The user can also specify that the string that will be scanned should contain both 1- and 2-byte character codes with special mode characters indicating which mode (1- or 2-byte) should be used to interpret the string. An optional escape can be requested on 1-byte values less than hex 40.

Search: This instruction searches through the elements of an array, checking specified portions of each element for the first occurrence of a specified value. The array can be defined as a character array or as a numeric array. The elements of the numeric array are treated like character strings. The relative index value for each element that contains a match is placed in the receiver operand. If the receiver is a data element, only the first element containing a match is noted. If the receiver is a data array, multiple occurrences of matches can be noted.

Verify: This instruction searches the source string to ensure that it is composed solely of characters from a chosen set. The relative location of the first character of the source that is not from the chosen set is returned in the receiver; otherwise, a zero value is returned. If the locations of multiple occurrences of characters not from the chosen set are desired, the receiver must be a binary array.

Translate With Table: This instruction copies selected bytes from a table to a receiver string. Instruction execution begins with the leftmost byte of a source string and proceeds byte-by-byte, left-to-right. The machine locates the selected byte in the table by adding the value in a source string byte to the table location. The byte from the table is copied to the receiver string at the same relative location as the source string byte.

Trim Length: This instruction determines what the resultant length (in bytes) of a source string would be if the bytes with a specific hexadecimal value were trimmed from the rightmost end of the source string.

Instruction execution begins with the rightmost byte of the source string and proceeds byte-by-byte, right-to-left. Each byte of the source string is compared to a specific trim value. If the two values are equal, the resultant length value of the source string is reduced by 1.

Instruction execution ends when the two compare values are unequal or the length value is reduced to 0.

Note: This instruction does not change the values in the source string bytes or the trim byte.

BOOLEAN OPERATIONS

The Boolean instructions operate on strings of characters. They perform their functions logically bit by bit on the corresponding bits of each of the source strings and the receiver string.

Boolean Instructions

System/38 supports the AND, OR, EXCLUSIVE OR (XOR), and NOT Boolean (or logical) instructions.

The source bit and the result bit value for each Boolean instruction is summarized as follows:

Source Bit Value		Result Bit Value			
OPERAND 1	OPERAND 2	AND	OR	XOR	NOT ¹
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

¹NOT refers only to source operand 1.

COMPARISON OPERATIONS

The Compare instructions test the relationship of the value of data items to each other and perform some action based on the results of the test.

The extender field and one or more branch or indicator operands define the branch or indicator options and the branch or indicator locations, respectively.

The compare instructions support three types of comparison:

- Arithmetic comparison is a sign and magnitude test between corresponding numeric scalars.
- Byte comparison is a byte-by-byte test between corresponding character or numeric scalars in which each byte is treated as an unsigned 8-bit binary value. Left or right adjusting of the operands along with right or left padding is optional before comparison.
- Bit comparison is a bit-by-bit test of selected bits of a 1-byte string scalar (numeric or character) as indicated by a byte string mask operand.

Comparison Instructions

The following System/38 instructions perform comparisons.

Compare Numeric Value: This instruction performs an arithmetic comparison of two numeric operands. The operation code specifies whether the branch or indicator option is to be used. The extender field specifies the conditions for which branches or indicator assignments are made.

Compare Bytes: Compare byte instructions perform logical comparisons. The various instructions provide the following options on the alignment of string operands prior to the comparison.

- Left-adjusted with no padding—The comparison considers the leftmost bytes of each operand. The length of the operation is equal to the number of bytes in the shorter operand.
- Left-adjusted with padding—The shorter operand is logically padded (on the right) to the length of the longer operand. The length of the operation is equal to the number of bytes in the longer operand.
- Right-adjusted with no padding—The comparison considers the rightmost bytes of each operand. The length of the operation is equal to the number of bytes in the shorter operand.
- Right-adjusted with padding—The shorter operand is logically padded (on the left) to the length of the longer operand. The length of the operation is equal to the number of bytes in the longer operand.

The operation code specifies whether the branch or indicator option is to be used. The extender field specifies the conditions for which branches or indicator assignments are made.

Test Bits under Mask: This instruction checks specific bits in the leftmost byte of the source operand to see whether they are binary 1 or binary 0. The bits to be checked are indicated by a 1 in the corresponding bit of the mask operand. The comparison is performed bit by bit. The operation code specifies either the branch or the indicator option is to be used. The extender field specifies the conditions for which branches or indicator assignments are made.

OBJECT MOVEMENT AND CONVERSION OPERATIONS

The movement and conversion instructions move a source operand to a receiver operand. These instructions move hex digit values, numeric values, byte strings, and character strings.

Byte string movement instructions operate on either character or numeric operands. The operation is a logical operation with no value conversion implied. The length of the operation is determined from the operands. Implicit or explicit padding or truncation is performed as described with the instruction.

Movement Instructions

The following instructions perform the movement of values.

Copy Numeric Value: This instruction copies the numeric value of an operand to another operand and provides any type of data conversion and precision adjustment needed.

Copy Operation: The copy operation copies logical byte strings from one operand to another. Four instructions provide the following options for the copy operation:

- **Copy Bytes Left-Adjusted with No Padding**—The leftmost bytes of the source operand are copied to the leftmost bytes of the receiver. The length of the operation is equal to the shorter of the two operands.
- **Copy Bytes Left-Adjusted with Padding**—The leftmost bytes of the source operand are copied to the leftmost bytes in the receiver. The length of the operation is equal to the number of bytes in the receiver. If the receiver is longer than the source, the excess bytes in the receiver (at the right) are assigned a value equal to the pad value.

- **Copy Bytes Right-Adjusted with No Padding**—The rightmost bytes of the source operand are copied to the rightmost bytes of the receiver. The length of the operation is equal to the length of the shorter of the two operands.
- **Copy Bytes Right-Adjusted with Padding**—The rightmost bytes of the source operand are copied to the receiver. The length of the operation is equal to the length of the receiver. If the receiver is longer than the source, the excess bytes in the receiver (at the left) are assigned a value equal to the pad value.

Copy Bytes with Pointers: This instruction copies both scalars and pointers from the byte string specified by the source operand to the byte string specified by the receiver operand. The validity of the pointers in the source operand is maintained when copied into the receiver operand.

Copy Bytes Repeatedly: This instruction performs the same function as the Copy Bytes Left-Adjusted instruction except with the additional function of the source operand being repeatedly copied until the receiver operand is filled.

Copy Bytes Overlap: This instruction performs the same functions as the Copy Bytes instruction with the additional feature of having predictable results should the two operands be unaligned and overlapping. The various instructions have the following two options for this operation: left-adjusted (no padding) and left-adjusted (with padding).

Exchange Bytes: This instruction exchanges the logical character string value of two equal length operands.

Copy Hex Digit: This instruction has four versions that can copy a specific 4-bit hex digit from the leftmost byte of the source operand to a specific 4-bit portion in the leftmost byte in the receiver operand. This instruction provides the following copy options:

- Numeric to numeric
- Numeric to zone
- Zone to numeric
- Zone to zone

The zone portion of a byte is defined as the leftmost 4 bits (bits 0-3); the numeric portion of a byte is defined as the rightmost 4 bits (bits 4-7).

Copy Bits with Left Logical Shift: This instruction copies the bit string value of the source operand to the bit string defined by the receiver operand. A left logical shift of the source string occurs under control of the shift control operand.

Copy Bits with Right Logical Shift: This instruction copies the bit string value of the source operand to the bit string defined by the receiver operand. A right logical shift of the source string occurs under control of the shift control operand.

Conversion Instructions

The following instructions perform explicit conversion.

Convert Character to Hex: This instruction converts the characters of the source operand into hex digits and places them in the receiver operand. The following are valid characters that can be converted to hex digits:

From Characters	To Hex Digits
Hex F0–hex F9	Hex 0–hex 9
Hex C1–hex C6	Hex A–hex F

An attempt to convert any characters other than these signals a conversion exception.

Convert Hex to Character: This instruction converts the hex digits of the source operand into characters and places them in the receiver operand.

Convert External Form to Numeric Value: This instruction scans a character string that represents a valid decimal number in display format, removes the display characters, and places the resulting zoned decimal scalar in the receiver operand.

Convert Character to Numeric: This instruction treats the character string source operand as if it were the numeric scalar described by the attributes operand, and places the result in the receiver operand. If the character string source operand is identified as zoned, a blank (hex 40) is allowed for the sign byte.

Convert Numeric to Character: This instruction converts the numeric value of the source operand to the type indicated in the attributes operand and places the result in the receiver operand.

Convert Character to BSC: This instruction uses the data compression technique in order to decrease the length of a character string.

Data compression occurs as the source string operand is copied to the receiver string operand. Any string of three or more blanks (hex 40) in the source string is not copied, but it is replaced by a 2-byte blank compression entry in the receiver string.

The source string is not changed by this instruction.

Convert BSC to Character: This instruction:

- Copies a source string to a receiver string
- Detects the presence of blank compression entries in the source string
- Interprets the blank compression entries
- Adds the correct number of blanks (hex 40) in the appropriate location of the receiver string

The characters associated with the blank compression entries are not copied to the receiver string.

The source string is not changed by this instruction.

Note: The format of the source string should be the same as that produced by the Convert Character to BSC instruction.

Convert Character to MRJE: This instruction uses the data compression technique in order to decrease the length of a character string.

The data compression options are either full compression or truncate trailing blanks.

Data compression occurs as the source string operand is copied to the receiver operand.

When the full compression option is specified, the bytes of the source string are interrogated to locate the blank character strings (two or more consecutive blanks), identical character strings (three or more consecutive identical characters), and nonidentical character strings in the source.

When the truncate trailing blanks option is specified, the bytes of the source string are interrogated to determine if a blank character string exists at the end of the source string. If one exists, those characters prior to it are treated as one string of nonidentical characters.

The system builds a string control byte in the receiver to describe each encountered string.

The source string is not changed by this instruction.

Convert MRJE to Character: This instruction decompresses a source string and puts the result in a receiver string. To accomplish decompression, this instruction:

- Detects the presence of string control byte entries in the source string
- Interprets the string control bytes
- Builds a receiver string as determined by the string control bytes

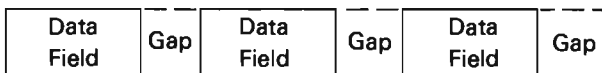
The source string is not changed by this instruction.

Note: The format of the source string should be the same as that produced by the Convert Character to MRJE instruction.

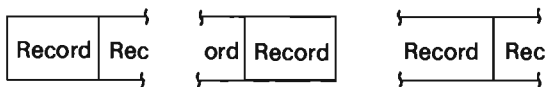
Convert Character to SNA: This instruction converts source data from character to the SNA (systems network architecture) format.

The source data is described (through certain controlling data) as being composed of one or more fixed length fields. The data fields can be separated by fixed length gaps of characters which are ignored during the conversion operation.

Source data:



Data processed as source records:



The source data can be specified as a single string of data or as a group of data records.

The optional functions that can be performed on the source data are trailing blank truncation, data transparency, and compression. An algorithm modifier specifies the optional functions to be performed.

Convert SNA to Character: This instruction converts source data from the SNA (systems network architecture) format to character.

The conversion operation is performed either on a record-by-record or on a string basis and is determined by the functions selected in the controlling data.

The optional functions that can be selected are record separator conversion, decompression, data transparency conversion, and blank padding.

Note: The format of the source data should be the same as that produced by the Convert Character to SNA instruction.

Cipher: This instruction uses the ANSI (American National Standards Institute) DEA (Data Encryption Algorithm) to encipher or decipher the data in the source operand and to place it in the receiver.

Cipher Key: This instruction uses the ANSI DEA to optionally perform one of the following:

- Generate a random key and place the result in the receiver
- Decipher a source cipher key
- Encipher the cipher key value and place the resulting key value in the receiver
- Verify the master key and return a verification code in the receiver
- Generate a PIN (Personal Identification Number) and return the result in the receiver
- Verify a PIN
- Translate a PIN and return the result in the receiver

BRANCHING OPERATIONS

Branching is the means of altering the sequential flow of instruction execution. The instructions in a program are numbered sequentially beginning with 1 for the first instruction. The operand of a branch instruction must identify the instruction number of the target instruction. The number of the target instruction must be within the instruction stream containing the branch instruction.

An operand that is designated as a branch target can be one of the following:

- Instruction number—A signed immediate operand indicating the number of the target instruction.
- Relative instruction number—A signed immediate operand indicating the displacement (in instructions) to the target instruction from the instruction being executed.
- Branch point—An object defined in the ODT with a value indicating the target instruction number.
- Instruction pointer—An object defined in the ODT serving as a variable pointer containing instruction numbers that can be altered during execution.

There are two kinds of branching: unconditional and conditional.

Unconditional Branching

The Branch instruction unconditionally transfers control to the instruction indicated in the branch target operand. The branch operand can be an immediate (absolute or relative) instruction number, a branch point, an instruction pointer, or an element of an IDL (instruction definition list).

If the IDL element is specified, the Branch instruction unconditionally transfers control to one of the instructions indicated in an IDL. The IDL is defined in the ODT as a list of branch targets with each one identifying an instruction within the instruction stream containing the IDL. Each entry in the IDL can be an instruction number of a branch point. The entry selected for use in the branch is determined by the value of the index suboperand in a subscript compound operand in the Branch instruction.

For example, assume the following declarations:

```
DCL A  BRANCH POINT VALUE (4)
DCL C  BRANCH POINT VALUE (7)
DCL X  BIN (2)
DCL TABLE IDL (A,5,17,C)
```

Executing the following instructions results in an unconditional branch to the indicated instruction numbers:

- CPY NV X,3
 B TABLE,X (Instruction number 17)
- B TABLE,4 (Instruction number 7)

Conditional Branching

Conditional branching can be specified for instructions that allow the optional branch form.

These instructions use branch targets as operands. They conditionally transfer control to the instruction indicated by the branch target based on the status of the instruction execution. Branch points, instruction pointers, instruction numbers, relative instruction numbers, or elements of an IDL can be specified as branch targets.

Up to four mutually exclusive resultant conditions are defined by these instructions (for example: high, low, or equal). Each instruction that allows conditional branching has a set of resultant conditions. If conditional branching is desired, these instructions include the extender field and one or more branch target operands in addition to the normal operands. The extender field specifies the instruction status branch options for which branches are to occur; the branch target operands specify the branch locations.

Each resultant condition can be ignored or associated with one of the branch target operands so that the branch occurs regardless of whether the condition does or does not occur.

A single branch operand can be associated with the presence of a given condition or the absence of this condition. A branch operand can be associated, for example, with an equal condition or with a not equal condition (implying high or low).

The extender field is composed of four 4-bit fields, each of which defines a branch option for which a branch may occur. Each branch option subfield corresponds to a branch target. After the basic instruction is executed, the branch options are compared with the resultant condition. Based on the result of the comparison, a branch may be executed to the corresponding branch target.

Variable Branching

A branch target operand can cause control to be passed to one of several locations based on the previous execution of the program. This can be accomplished by either using the Branch Indexed instruction as previously described, or using an instruction pointer as the target of a Branch instruction or as one of the target operands for the branch form of an instruction.

The instruction pointer can be given a value with a Set Instruction Pointer instruction and can then be referred to as a branch target. The Set Instruction Pointer instruction can be set to a branch point, an instruction number, or a relative instruction number (relative to the Set instruction).

EDITING OPERATIONS

Editing is the process of changing the value of a scalar from an internal form to an EBCDIC form that is more suitable for display on an output device.

Editing Instructions

Edit: The Edit instruction performs the following editing functions while transforming the source operand to the receiver operand:

- Unconditional insertion of a source value digit with a zone as a function of the source value's algebraic sign
- Unconditional insertion of a mask operand character string
- Conditional insertion of one of two possible mask operand character strings as a function of the source value's algebraic sign
- Conditional insertion of a source value digit or a mask operand replacement character as a function of a source value leading-zero suppression
- Conditional insertion of either a mask operand character string or a series of replacement characters as a function of a source value leading-zero suppression
- Conditional floating insertion of one of two possible mask operand character strings as a function of both the source value algebraic sign and leading-zero suppression

Test and Replace Characters: The Test and Replace Characters instruction scans a character string in a left-to-right manner. All characters encountered prior to the first nonzero zoned decimal digit are replaced with a specified fill character.

LOGICAL CHARACTER OPERATIONS

A Logical Character instruction performs unsigned binary arithmetic functions. The operands added by the Add Logical Character instruction or subtracted by the Subtract Logical Character instruction must be character scalars. The length of the operands can be a maximum of 256 bytes.

ARRAY INDEX OPERATIONS

Single dimensioned arrays are supported. The CAI (Compute Array Index) instruction can be used to reduce the subscript value of a multidimensional array to a single index value. This single index value can be used to address a single dimensioned array.

Following is an example of the use of the CAI instruction. Assume that in a high-level language a three-dimensional array is declared with five elements in the first dimension, four in the second, and three in the third.

```
DCL A(5,4,3)
```

This is equivalent to a declare of a single dimension array of 60 elements.

```
DCL A(60)
```

If A(I,J,K) is to be located, where I=4, J=2, and K=3, the instructions that would be used are:

```
CAI X,I,J,5      X=4+(2-1)*5 = 9
CAI X,X,K,20     X=9+(3-1)*20 = 49
```

A reference to A(X) locates the 49th element of the array.

Note: The operands on the Compute Array Index instruction are restricted to a BINARY(2) element.

NO OPERATION

The No Operation and the No Operation And Skip instructions do not perform any function. They do, however, insert gaps in the instruction stream, thereby preventing adjacent instruction addresses (numbers) from being physically adjacent.

These instructions are not assigned an instruction number. Therefore, they cannot be the target of a branch instruction. Additionally, these instructions are not counted as instructions in the instruction stream.

The No Operation and the No Operation and Skip instructions can precede or follow any System/38 instruction and can be repeated an unlimited number of times in succession.

Program Execution

Program execution causes the instructions in the encapsulated program to be performed within a process. These instructions operate against the objects defined in the program and any secondary objects referenced by these objects.

Program execution consists of two logically distinct operations:

1. Activating the program causes the static storage for the program to be allocated and initialized within the process.
2. Invoking the program causes the automatic storage defined in the program to be allocated and initialized. This operation also causes control to be passed to the external entry point of the program.

PROGRAM ACTIVATION

Program activation makes the program ready for use by the activating process. The result is an activation entry through which values saved across program invocations can be stored, and through which references can be made to the program by other programs within the process.

A program can be explicitly activated by the Activate Program instruction, or implicitly activated by referencing that program for invocation if it has not already been activated within the process.

A program is always activated before it is invoked. However, programs that require no static storage are considered to be permanently activated and do not require activation prior to invocation.

Activation Creation

When a program is activated, an area in the PSSA (process static storage area) of the process is allocated to contain the program's static storage. The contents of this static storage area is then available each time the program is invoked within the process. This storage contains the static objects for the program, a space pointer to the next activation entry (if one exists) in the PSSA, and attributes specifying the status of the activation. Objects that do not have program-defined initial values can optionally be given the system default initial value (hex 00). This option is selected through use of a program attribute on the create program template.

Note: An improvement in performance might be realized if the PSSA is not initialized by default, even though the initial values specified for individual data objects are still set.

Activating a program also makes the external data objects that are defined in the program available to other programs in the process.

Each activation entry in the PSSA has the following format (see Figure 3-1):

- A space pointer that locates the previous activation entry (the first activation entry locates the PSSA header)
- A space pointer that locates the next activation entry (unchanged for the last activation entry)
- A system pointer that locates the associated program
- A number that identifies the activation in the chain
- An activation status bit that indicates whether the activation is currently active
- An invocation count that indicates how many invocations are currently invoked and using this activation
- An activation mark that indicates the relative time during which the activation was activated

- A number that specifies the total length of the activation
- Static storage for the program

The PSSA is located by a space pointer that was specified when the process was initiated. The location identified by the space pointer is considered to be the beginning of the PSSA and must be 16-byte aligned. At this location is a 96-byte PSSA base entry that consists of the following:

- A space pointer that locates the last activation entry in the process (addresses the base entry if no programs are activated)
- A space pointer that locates the first activation entry in the process (ignored if no programs are activated)
- A space pointer that locates the next available storage location in the PSSA space
- PSSA chain control bits that are used when programs explicitly alter the status of the PSSA chain with direct bit manipulation

The user must initialize this PSSA base entry before the first program is activated in the process. (The Materialize Process Attributes instruction can be issued to locate the PSSA.)

When a program is activated, an attempt is made to allocate enough space in the PSSA to contain the activation entry. If the PSSA space is extendable and is currently not large enough to contain the entry, the PSSA is implicitly extended by the machine. If the PSSA is fixed in size or cannot be extended to contain the entry, an exception is signaled.

The following occurs when the new activation entry is initialized:

- The previous activation entry pointer is set to address the most current activation entry.
- The associated program pointer is set to address the activated program.
- The activation is marked as active.
- Static storage for the program is initialized as defined in the program definition.

- The invocation count is set to zero.
- The activation mark is obtained by incrementing the mark counter value (which is carried internally in the machine) by one and copying the resulting value.
- The length field is set to the number of bytes of storage occupied by the PSSA header and the static data following it.

The next available storage location in the PSSA base entry is set to address the next available 16-byte aligned area beyond the new activation entry.

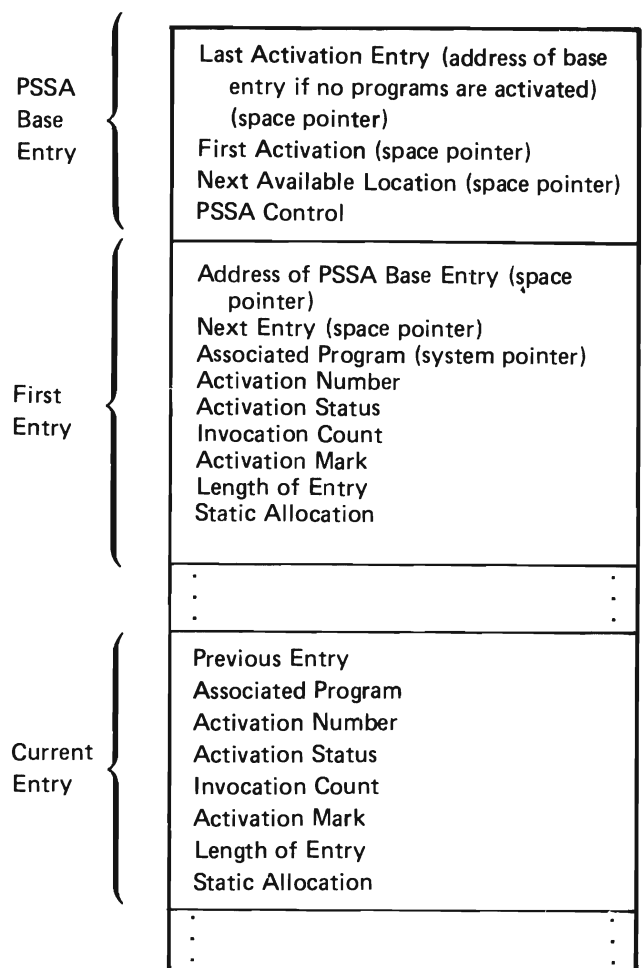


Figure 3-1. Process Static Area Structure

If the activation of a program already exists within the process chain and is active when the Activate Program instruction is executed, the static storage of the program is reused. (The static storage of the program may or may not be reused if the activation is inactive.) In either case (active or inactive), the static storage of the activation is reinitialized, the activation is set to the active state, and a space pointer is set to address the reinitialized activation.

When a new activation is allocated, the space that is occupied by inactive activations may be used for the new activation. All PSSA entries (activations) that are inactive, have an invocation count of 0, and appear as the last entries in the linked PSSA chain may be removed from the PSSA chain. The space occupied by them can then be used to allocate new activations.

The new activation is allocated space starting at the lowest possible address in the PSSA. However, this address must be higher than the address of any active activation in the chain.

If no activations remain (after being removed under the previous conditions), the new activation is placed at the lowest address of the removed activations. Note that in the previous cases the next available location pointer in the PSSA base entry is not used to determine the address at which to allocate a new activation. If the last activation in the PSSA chain is not removed at the beginning of execution of the Activate Program instruction, then the next available location pointer in the PSSA base entry specifies the location at which the new activation is to be allocated.

If the program to be activated does not require static storage, no activation entry is allocated.

Values of static objects in the activation can be modified. These values remain modified even though the activation may not be currently invoked.

Note: The static storage area of a process must be modified by only that process. Otherwise, the process may produce unpredictable results.

Activation Destruction

The Deactivate Program instruction can designate an activation entry as not active. When this occurs, an activation entry must be reactivated before it can be invoked. The machine implicitly reactivates a not active entry when the associated program is to be invoked.

Only those activations with a zero invocation count can be de-activated. A program can de-activate itself if it is the only invocation of that program in the process.

PROGRAM INVOCATION

The program invocation functions control synchronous execution within a process. These functions allow control to pass from one program instruction stream to another and also allow for a subsequent return of control when a function is complete.

Invocation Creation

When a program is invoked, the following occurs:

- The execution of the invoking program is suspended, and the current status is saved pending return of control.
- An invocation entry for the invoked program is allocated in the PASA (process automatic storage area). This entry contains an allocation for each object that has the direct on automatic allocation attribute.

- The automatic objects are assigned initial values as follows:
 - Objects with program-defined initial values are given those values.
 - Objects that do not have program-defined initial values can be optionally given the system default initial value (hex 00). This option is selected through use of a program attribute on the create program template.

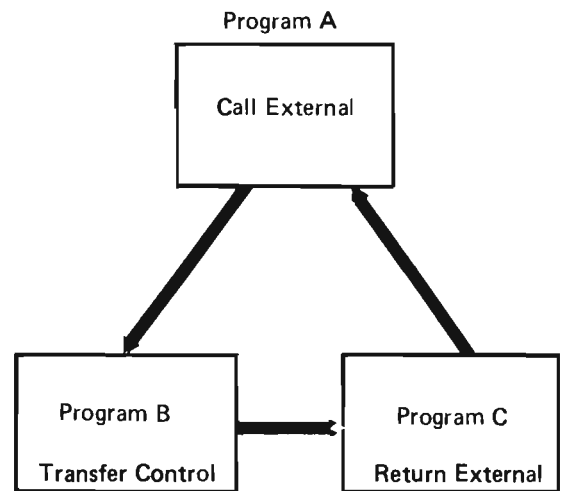
Note: An improvement in performance might be realized if the PASA is not initialized by default, even though the initial values specified for individual data objects are still set.

- Exception descriptions defined in the program become active and thereby exercise control over exception handling functions for the associated exceptions.
- Parameter objects defined in the invoked program are resolved to argument objects passed by the invoking program.
- If the referenced program specifies the adopt user profile attribute, the program owner (user profile) is used to supplement the authority available to the process. This authority can be propagated to subsequent invocations if the program also specifies the propagate adopted user profile attribute.
- The program to be invoked receives control at its external entry point. Instruction execution is sequential except where modified by a branching instruction, an instruction that causes control to pass from the invocation (for example, a Call External instruction or a Return External instruction), exception handling, or event handling.

Programs can be invoked in the following instances:

- During the process initiation phase, a process definition can specify a program to be invoked.
- A process definition must specify the first program to be invoked in the problem phase. When process initiation enters the problem phase, this program is given control.

- A Call External instruction suspends execution of the invoking program and passes control to the referenced program.
- A Transfer Control instruction suspends execution of the invoking program and deallocates the invocation of that program; control is then passed to the referenced program. When control returns from the referenced program, execution resumes in the invocation that immediately preceded the transferring invocation in the process hierarchy. For example, in the following illustration, program A executes a Call External instruction to program B; program B executes a Transfer Control instruction to program C; program C then executes a Return External instruction thereby causing control to be returned implicitly to program A.



- An exception description can specify a program to be invoked when a specified exception occurs.
- An event monitor can specify a program to be invoked when a specified event occurs.
- A process definition can specify a program to be invoked as part of a process phase termination.
- An invocation exit program can be invoked if the requesting invocation is bypassed because of normal exception handling actions or because of process termination.

Each invocation entry in the PASA has the following format:

- Space pointer locating the previous invocation entry (the first invocation entry addresses the PASA header).
- Space pointer locating the next invocation entry (not modified for the most current invocation entry).
- System pointer to the associated program.
- Invocation attributes. A value that specifies the number of the invocation relative to the number of programs currently invoked in the process. The first invocation entry is number one. The type of invocation is also provided to indicate how the program was invoked.
- Invocation mark. A number that indicates the relative time of invocation of the program.
- Program's automatic storage.

The update PASA stack program attribute (which was specified on program creation) indicates whether or not the program requires that the PASA stack information, which is contained in both the PASA base entry and the invocation entries, be updated.

Upon invocation of a program that requires the stack be updated, it is possible that prior invocations may exist that did not require the stack update. These prior invocations would not have their associated stack information updated to reflect the current chain of invocations active in the PASA. If necessary, the PASA stack information in the PASA base entry and the prior invocation entries are updated with the current status prior to continuing with the invocation of a program requiring update of the PASA stack.

The PASA is located by a space pointer that was specified when the process was initiated. The location identified by the space pointer is the beginning of the PASA and must be 16-byte aligned. The 96-byte PASA base entry consists of the following (see Figure 3-2):

- A space pointer that locates the current invocation entry in the process. (If no programs are invoked, this pointer must address the PASA base entry.)
- A space pointer that locates the first invocation entry in the process (ignored if no programs are invoked).
- A space pointer that locates the next available storage location in the space containing the PASA.
- A mark counter that is incremented for each activation and invocation to indicate the relative time of their allocation.

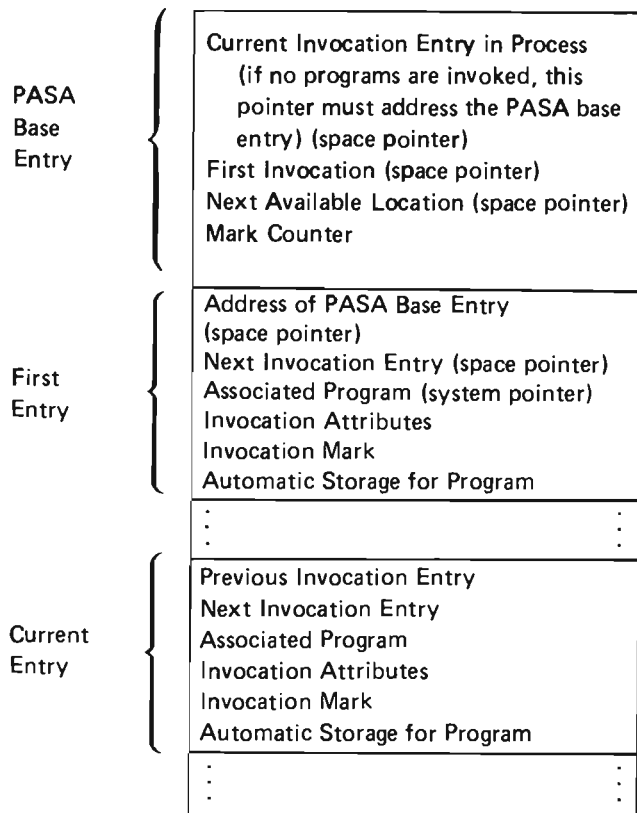


Figure 3-2. Process Automatic Storage Area Structure

The PASA base entry must be initialized by the user before the process is initiated. (The user can issue the Materialize Process Attributes instruction to locate the PASA.) The current invocation entry in process, next available storage location, and mark counter values are accessed as input to the machine only during the initiation of the process. Thereafter, the machine maintains these values internally. The PASA base entry fields are optionally updated on each program invocation. This update is dependent upon whether or not the program being invoked has specified the update PASA stack program attribute.

A space pointer that locates the PASA invocation entry for the currently executing program can be materialized by the Materialize Invocation Entry instruction.

When a program is invoked, enough space is allocated within the PASA to contain the invocation entry. When the initial program in the process is invoked, the space used for the allocation is located by the next available storage location pointer in the PASA base entry. For all other invocations of programs within the process, the space used for the allocation is located by an internal machine value which is maintained with the space address of the next available storage location. This value must address a 16-byte aligned area in the space or a boundary alignment exception is signaled.

When the space is currently not large enough to contain the entry and the size of the space is extendable, it is implicitly extended. If the size (length) of the space is fixed or the size cannot be extended enough to contain the entry, an exception is signaled.

When a program is created with the update PASA stack attribute and the program requires that the PASA stack be updated, the new invocation entry is updated as follows:

- The previous invocation entry pointer is set from the current invocation entry in process address value. This value is carried internally and locates the calling invocation entry.
- The next invocation entry is not modified.
- The associated program pointer is copied from the associated activation or the operand 1 system pointer if no activation exists.
- The invocation type value is set to indicate how the program was invoked.
- The value of the mark counter (which is carried internally) is incremented by one and the new value is copied to the invocation mark field. The new value is also copied to the activation mark field of the program's activation if the activation was initialized by this instruction.
- The user area field is set to binary zero.
- The program's automatic storage is initialized as defined in the program definition.

When a program is created with the update PASA stack attribute and the program does not require that the PASA stack be updated, the new invocation entry is updated as follows:

- The value of the mark counter (which is carried internally) is incremented by one. The new value is copied to the activation mark field of the program's activation if the activation was initialized by this instruction.
- The PASA stack information that is necessary for subsequent program invocations or updating of stack information for this invocation is stored internally. This includes values associated with this invocation for the previous invocation entry address, next available storage location, program pointer, invocation number, invocation type, and mark counter.
- The program's automatic storage is initialized as defined in the program definition.

When a program is created with the update PASA stack attribute and the program requires that the PASA stack be updated, a space pointer addressing the new invocation entry is stored in the next invocation entry pointer of the invoking invocation.

When a program is created with the update PASA stack attribute and the program requires that the PASA stack be updated, a space pointer addressing the new invocation entry is stored in the current invocation entry pointer of the PASA base entry. The next available storage location in the PASA base entry is set to address the next available 16-byte aligned area beyond the new invocation entry.

The PASA entry created for a program with no automatic data consists only of a stack control entry.

After the static and automatic storage allocation and/or initialization function is completed, control is passed to the invoked program.

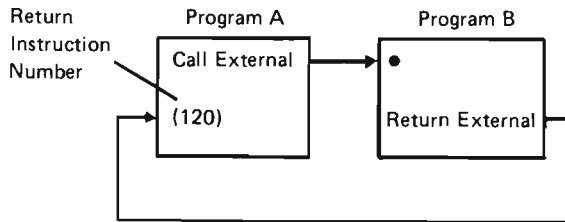
Invocation Destruction

When an invoked program yields control to another program, its invocation is deallocated with the following results:

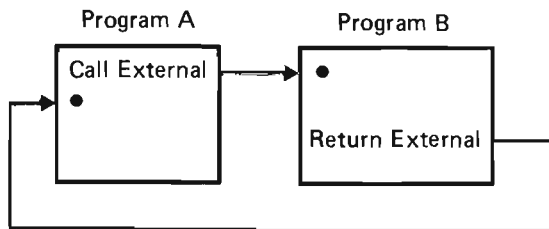
- Execution of the invoked program is suspended.
- Automatic objects are deallocated. The invocation entry is removed from the PASA chained list. The current invocation entry in process entry in the PASA base entry is set to address the immediately preceding invocation and addressability to the current invocation is set into the next available storage location entry in the PASA header. Storage in the space containing the PASA is not deallocated and is available for allocation for future invocations.
- Exception descriptions associated with the invocation no longer exercise control over exception handling functions.
- Depending on the mechanism used to destroy the invocation, control is passed to another program executing in the process, or the process is destroyed if there is no higher level invocation.

An invocation may give up control and subsequently be destroyed with a Return External instruction, Transfer Control instruction, or with exception returns.

With a Return External instruction, the invocation is destroyed, and control passes to the immediately preceding invocation. If the invoking Call External instruction (in the preceding invocation) specified a return list, a value can be specified in the Return External instruction to select a return target in the preceding invocation.



If no return target is specified, control returns to the instruction immediately following the invoking Call External instruction.



If the Return External instruction is executed in the highest invocation in either the process initiation phase or the process problem phase, the next phase of the process is entered. If the instruction is executed in the highest invocation of the process termination phase, the process is terminated.

If the Return External instruction is executed in the highest invocation in an event handling sequence, control returns to the next sequential instruction in the process that experienced the event.

If the instruction is executed in an invocation of a program that caused a user profile to be adopted, the user profile is no longer used for authority verification in the process.

When a Transfer Control instruction is executed, the invocation of the program that contains the instruction is destroyed, and the referenced program is invoked.

Control is returned from a transferred-to invocation in the same manner as from the transferring invocation. The return point selection function refers to the immediately preceding invocation.

If the Transfer Control instruction is executed in an invocation of a program that caused a user profile to be adopted, the user profile is no longer used for authority verification in the process.

The Return from Exception instruction causes control to be returned to a specified invocation from an exception handling sequence. This instruction can only be executed during the initial invocation in the exception handling sequence. It causes that invocation to be destroyed. Control is passed to an invocation in the process related to the occurrence of the exception, the exception handling sequence, and the specific Return from Exception instruction.

Invocations can also be destroyed without a program executing a Return External, Transfer Control, or Return from Exception instruction. The exception management functions can cause an invocation to be destroyed when they give control to a higher level invocation to handle an exception or when they return control from an external exception handler to another program. Process termination causes implicit destruction of all invocations in the process.

Invocation Exit Programs

The Set Invocation Exit instruction establishes an invocation exit program that is to be given control if its associated invocation is destroyed because of normal exception handling actions or because of process termination. Normal exception handling actions are considered to be those actions that result from executing the Return From Exception instruction or the Signal Exception instruction.

The following occurs if any invocations are to be destroyed because of normal exception handling actions:

- An invocation exit program is given control if its associated invocation is to be destroyed. Exception management gives control to all such invocation exit programs before giving control to the exception handler or returning to the appropriate invocation.
- An event is signaled if the invocation to be destroyed is an invocation exit program. The exception handling in progress is aborted, the invocation exit program and its associated invocation are destroyed, and processing resumes with the exception handling action or with the process termination that caused this invocation exit program to be invoked in this process.

The following occurs if any invocations are to be destroyed when a process is terminated and the process is not in termination phase:

- An invocation exit program is given control if its associated invocation is to be destroyed.
- An event is signaled if the invocation to be destroyed is an invocation exit program. The invocation exit program and its associated invocation are destroyed, and process termination continues.

The invocation exit program established for an invocation is implicitly cleared when this program is given control.

The Clear Invocation Exit instruction disassociates the invocation exit program that was established for an invocation. No exception is signaled if an invocation exit program was not specified by its associated invocation.

Invocation Example

Figure 3-3 is an example of the invocation of programs within a process.

The flow of control throughout the process is indicated with numbered lines. These lines represent program execution functions performed within a process. The process is represented to be in one of three phases (initiation, problem, or termination). The indicated functions are allowed for all three phases.

- 1 Program A is invoked as the first invocation in the process.
- 2 Program A executes a Transfer Control instruction to pass control to program B. The invocation for program A is destroyed (but not the activation), and program B is invoked and given control.
- 3 Program B executes a Call External instruction to invoke program C.
- 4 An exception is signaled in program C, and the execution of program C is suspended in order to process the exception. The external exception handler, EX1, is invoked based on the associated exception description.
- 5 Program EX1 executes a Call External instruction to invoke program EX2.
- 6 Program EX2 executes a Return External instruction to cause its invocation to be destroyed and control to be passed to program EX1 at the instruction following the Call External.
- 7 The Return from Exception instruction is executed in program EX1. This causes its invocation to be destroyed and control returned to program C at an instruction based on the instruction address specified by the exception return.
- 8 Program C executes a Call External instruction to invoke program D.
- 9 Program D executes a Transfer Control instruction to pass control to program E. The invocation for program D is destroyed (but not the activation), and program E is invoked and given control.
- 10 Program E executes a Call External to invoke program F.
- 11 During the execution of program F, an event occurs that is monitored within the process. Execution of program F is suspended and the event handler, program EV1, is invoked.
- 12 A Return External instruction in the event handler is executed. The invocation for program EV1 is destroyed, and control is passed to the proper instruction in the suspended invoking program (program F).
- 13 Program F executes a Return External instruction. The invocation of Program F is destroyed, and control passes to the instruction following the Call External instruction in program E.
- 14 Program E executes a Return External instruction to pass control to the instruction following the Call External instruction in program C.
- 15 Program C executes a Return External instruction. The invocation of program C is destroyed, and control returns to the instruction following the Call External instruction in program B.
- 16 Program B executes a Return External instruction. The invocation of program B is destroyed and because it is the highest level invocation in the process, control returns to the process. If the process is in the initiation phase or the problem phase, execution continues in the next phase. If the process is in the termination phase, the process is terminated.

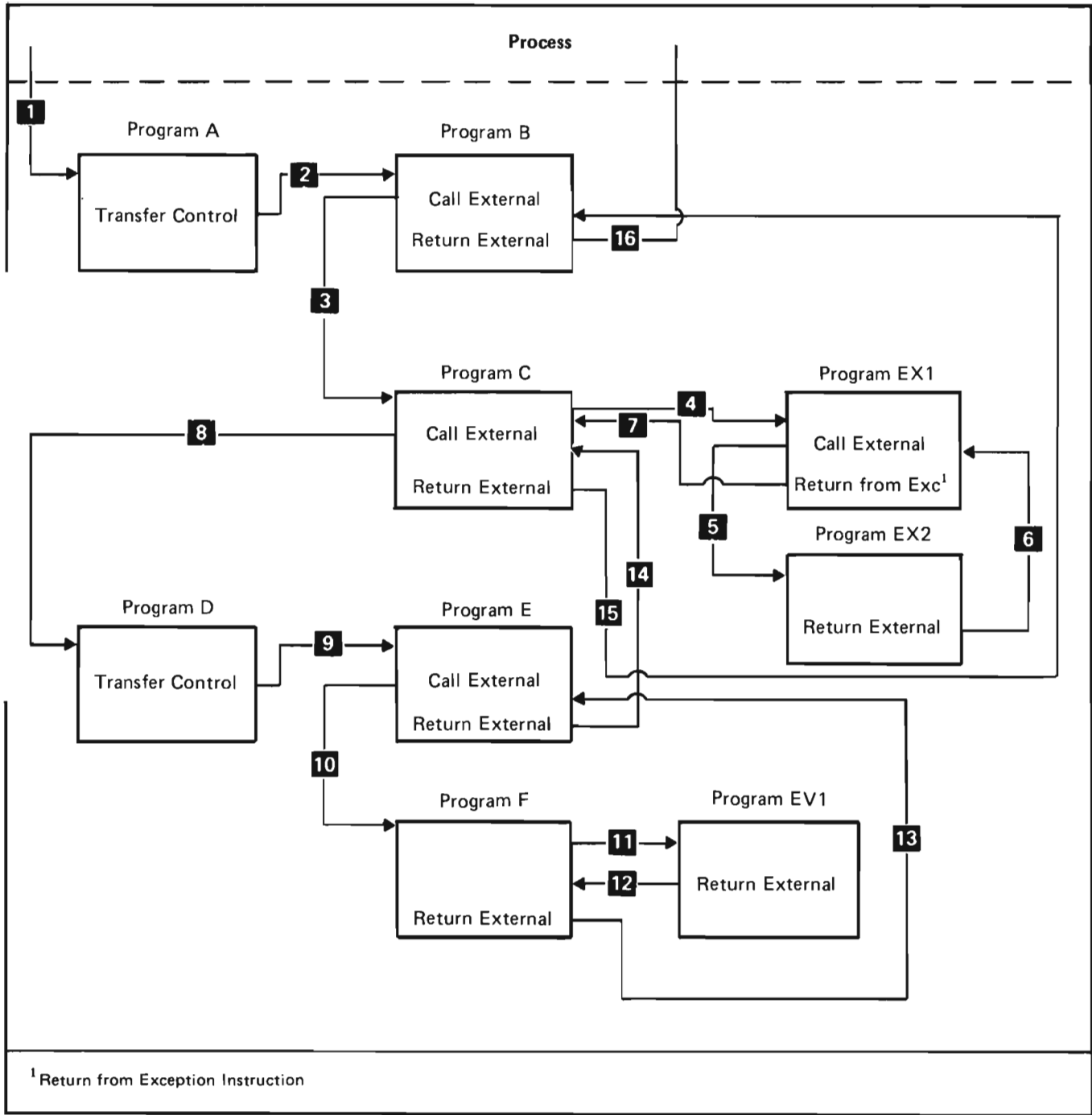
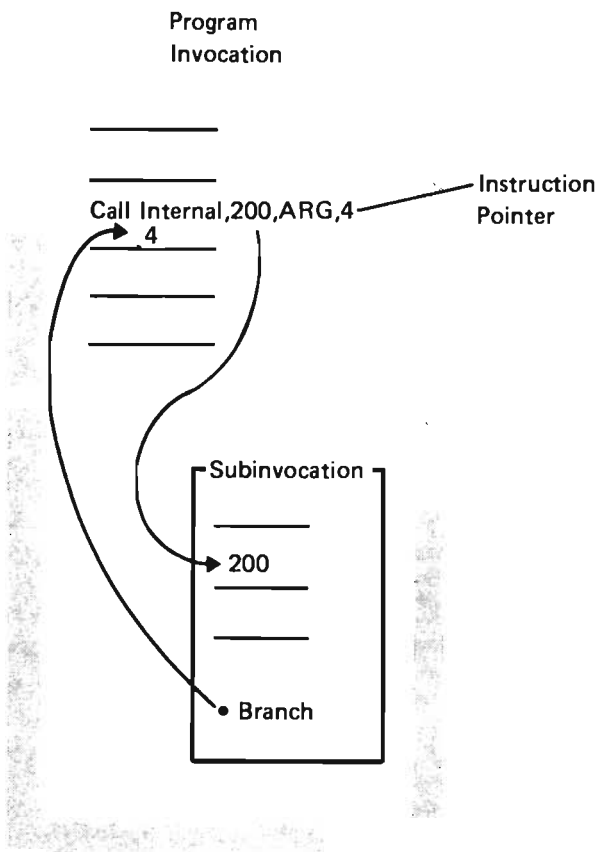


Figure 3-3. Program Invocation

Subinvocations

Subinvocations are internal invocations that occur within invocations of programs. This group of instructions can be specified only once in the instruction stream but can be executed more than once as a subinvocation.

A Call Internal instruction can invoke a subinvocation from any point in the program. A Call Internal instruction is actually a branch to an internal entry point with return linkage and optional arguments. The return linkage consists of setting an instruction pointer to address the instruction immediately following the Call Internal instruction.



No new automatic objects are allocated. A subinvocation has addressability to the same objects as the calling invocation.

The entry point target of the Call Internal instruction must be in the same instruction stream.

The subinvocation executes until it is terminated based on user specification. If desired, the instruction pointer set by the Call Internal instruction can be used as a branch operand to return to the previously executing instruction sequence; however, the machine does not enforce the return because a branch can be made to an alternate location.

The Call Internal instruction enables the calling invocation to bind one of several sets of arguments to the entry point parameter with each set of arguments corresponding to a different call. Simultaneously, the called invocation can use any object that is defined within the calling invocation.

After being set, the parameters remain bound to arguments until they are rebound with another Call Internal instruction or until the containing invocation is destroyed. A parameter can be associated with only one internal entry point in a program.

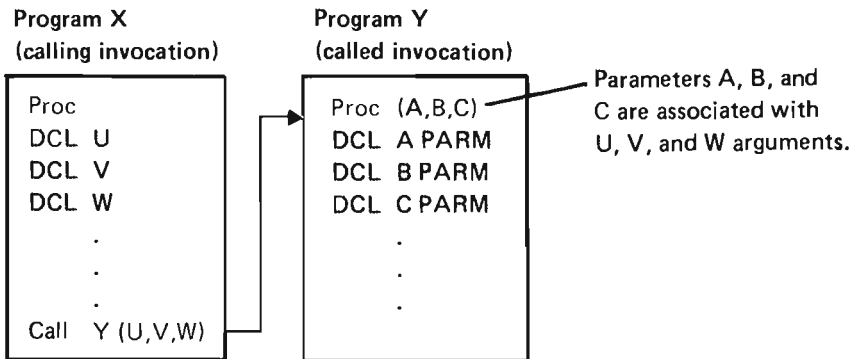
The Call Internal instruction without arguments does not provide processing of arguments because it uses the same objects as the caller.

When a Return External instruction is executed while a subinvocation is invoked, the entire invocation, including the subinvocations, is destroyed and control is passed as previously described.

Subinvocations can also be invoked through exception handling. If an exception occurs in an invocation and an exception description defined to handle that exception specifies an internal exception handler, a subinvocation is invoked. Control is passed to the internal entry point and execution continues until a Return from Exception instruction is executed. This instruction terminates the subinvocation and the exception handling sequence. Control is passed to the invocation and the instruction location within that invocation as specified by the instruction. The Return from Exception instruction is required to terminate the exception handling sequence.

ARGUMENTS AND PARAMETERS

Arguments and parameters provide a means of communicating information between two execution units. The information can be transferred from a creating process to a created process, from a calling invocation to a called invocation, from an invocation to a subinvocation, and between subinvocations of the same invocation.



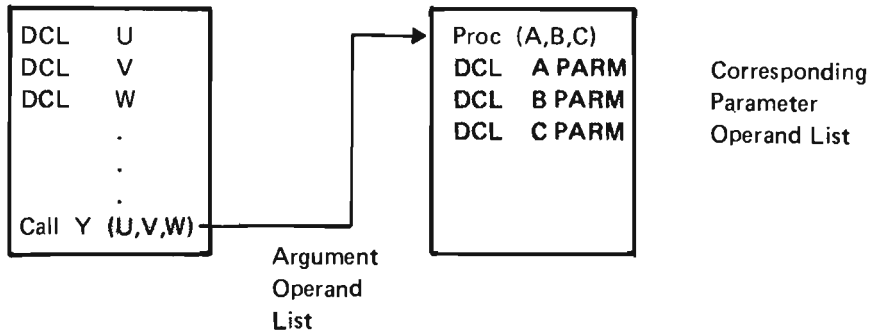
A parameter is mapped onto an argument during invocation. A parameter is not bound prior to invocation and is an indirect reference to an object provided by the invoking program.

The parameters associated with an entry point are referenced in a program object called an operand list (used as a parameter list). The operand list is, in turn, referenced by an entry point definition.

An argument is specified in an operand list (used as an argument list).

The operand list (used as an argument list) is a set of references to other program objects. It is the operand of a Call External or Transfer Control instruction and the mechanism whereby arguments (references to actual objects) are mapped to parameters (indirect references).

During invocation, the operand list being used as an argument list is referenced by the invoking instruction and matched with a corresponding operand list being used as a parameter list at the called entry point. For the duration of the invocation, references to the parameters in the invoked instructions refer to the corresponding arguments in the calling invocation.



The operational characteristics and functions vary slightly for interinvocation, intrainvocation, and interprocess communication. Because of this variance, they are presented in separate sections in the following text.

Interinvocation Communications

Arguments

The user specifies arguments as objects to be communicated to a succeeding invocation. Arguments can be used to specify input values to the called invocation or to receive output values that result from the execution of the called invocation.

Passing an argument implies that addressability to the argument and, therefore, the value contained in the argument is available to the succeeding invocation. Scalar and pointer data objects can be passed as arguments.

There is no special attribute to be specified in order for an object to be considered as an argument; a reference to an object in an argument list causes that object to be an argument.

Parameters

The user specifies parameters as objects to be received when the program is invoked. Parameters allow indirect references to arguments passed during invocation. This implies that addressability to an argument and, therefore, the value of the argument is available through a reference to a parameter.

A reference to a parameter is actually a reference to the argument. Because of this implied reference, parameters are given a special parameter attribute in the ODT. This attribute specifies the following:

- No storage is allocated for the parameter object when the program is invoked. A reference to the parameter gains addressability to the corresponding argument as defined at the time of the invocation.
- The machine does not validate any correspondence of argument type. The location addressed by the argument is referenced when the parameter is referenced. The attributes of the parameter view are used by the instruction.
- A reference to the parameter as a source operand causes the value of the corresponding argument to be used. A reference to a parameter as a target operand causes the value of the corresponding argument to be modified.

Scalar and pointer data objects can be defined as parameters.

Argument Lists

An argument list (an assumed type operand list) is an ordered list defining all of the arguments to be passed to a succeeding invocation or subinvocation on a Call External, Call Internal, or Transfer Control instruction.

The length of the argument list can be fixed or variable. A fixed-length argument list causes all of the objects defined in the list to be passed to the succeeding invocation. A variable-length argument list allows the program to determine the number of arguments to be passed from the list. The Set Argument List Length instruction changes the number of arguments to be passed. If the instruction specifies that *n* arguments are to be passed, a reference to that argument list on a Call External or Transfer Control instruction passes only the first *n* arguments in the list.

An initial length must be specified in the argument list definition. This length remains in effect until modified. Likewise, once the length is modified, it remains in effect until further modification. The modified length may vary from zero to the maximum size of the argument list.

The argument list definition must contain a reference to a program object for each possible element in the list. The only variability in the argument list is in the number of these references that are to be communicated to a succeeding invocation or in the values of the actual program object.

An argument list entry can reference any objects of the types specified in the definition. A reference is allowed to an object with a biased, defined, direct, static, automatic, or parameter attribute. A reference is not allowed to an array element or substring.

Multiple argument list entries can reference the same ODT object.

Addressability to an argument is established in the argument list during the execution of the Call External or Transfer Control instruction. The address of a variably addressed object can be changed prior to invocation but cannot be changed by the called invocation. For example, if a based data object and its base (a space pointer) were both passed as arguments, changing the value of the space pointer in the called invocation would not change the addressability of the parameter data object as known to the called invocation.

Note: Objects with the direct on automatic attribute cannot be passed as arguments by the Transfer Control instruction.

Parameter Lists

A parameter list (a parameter type operand list) is an ordered list that defines all of the parameters that are to receive information from the preceding invocation.

The length of a parameter list can be fixed or variable. A fixed-length parameter list specifies that the entry point expects to receive exactly the number of parameters defined. Otherwise, an exception is signaled. A variable-length parameter list specifies that the entry point expects to receive at least a minimum number of parameters but no more than the maximum number of parameters defined.

A fixed-length parameter list can receive arguments from a fixed-length or variable-length argument list as long as the actual number of arguments corresponds to the number the parameter list is expecting. Similarly, a variable-length parameter list may receive arguments from a fixed-length or variable-length argument list as long as the actual number of arguments is within the range of that allowed for the variable parameter list.

Parameter lists must be defined as internal or external. Internal parameter lists are referenced by internal entry points, and the lists must be fixed in length, as described under *Intrainvocation Communications* later in this chapter.

The parameter list definition must contain a reference to a program object for each possible element in the list. Each of the referenced objects must have the parameter attribute. Two entries in the same parameter list must not reference the same parameter object. A parameter must not be referenced in more than one parameter list (internal or external).

Argument/Parameter Correspondence

When an argument list is specified on a Call External or Transfer Control instruction, the individual arguments are intended to correspond one for one with the parameters in the parameter list. This correspondence includes the number of arguments and parameters but does not include the types of objects referenced as arguments and parameters.

The following illustration shows argument/parameter binding. Shown on the example are the high-level language form and the program template form of two programs, X and Y.

When program Y is invoked by program X, parameters A, B, and C in program Y are bound to corresponding arguments U, V, and W in program X. The Add Numeric instruction adds the contents of V and W and stores the sum in U.

High-Level Language Form		
Program X		Program Y
PROC		PROC(A,B,C)
DCL U		DCL A PARM
DCL V		DCL B PARM
DCL W		DCL C PARM
.		.
.		.
CALL Y(U,V,W)		A = B+C
.		.
.		.
.		.
END X		END Y

Program X

Object Definition Table	Instruction Stream
Entry Point X Instruction 1 External	<ul style="list-style-type: none"> • • CALL E,Z • • •
U Scalar Data Object	
V Scalar Data Object	
W Scalar Data Object	
E System Pointer-Initialize Program Y	
Z Operand List-Argument List (Fixed,3,U,V,W)	

Program Y

Object Definition Table	Instruction Stream
Entry Point Y Instruction 1 External Operand List (D)	<ul style="list-style-type: none"> • • ADDN A,B,C • •
A Scalar Data Object Parameter	
B Scalar Data Object Parameter	
C Scalar Data Object Parameter	
D Operand List-Parameter List (Fixed,3,A,B,C)	

Intravocation Communications

As with interinvocation communications associated with Call External and Transfer Control instructions, arguments can be passed on an internal call. A Call Internal instruction can reference an argument list that specifies the arguments to be passed. An internal entry point can reference an internal parameter list that specifies which parameters are to be received at the entry point.

Execution of a Call Internal instruction causes the parameters specified at the entry point to be associated with the arguments passed on the Call Internal instruction.

Argument lists and parameter lists for internal calls must be fixed in length.

A parameter cannot be referenced in both an internal and an external parameter list. However, a parameter can be referenced by more than one internal parameter list.

Once a parameter is associated with an argument, this association remains until one of the following conditions occurs:

- Another Call Internal instruction specifies an entry point that references the parameter. In this case, a parameter is once again associated with the argument.
- A Return External instruction is executed. In this case, all arguments and parameters are deallocated.

Interprocess Communications

Interprocess communications through the argument/parameter mechanism is supported by the Initiate Process instruction to the initial program in the problem phase of the process.

Argument and parameter functions associated with interprocess communications are as defined for interinvocation communications. As in interinvocation communications, only addressability is provided to the initiated process.



Chapter 4. Supervisor and Control Functions

Process Management

Process management supports the existence and management of multiple concurrent and asynchronous units of work in the machine.

A process controls the accomplishment of the units of work within the machine. Multiple asynchronous processes can exist concurrently and compete for the resources of the machine.

The resources allocated to processes include objects, auxiliary storage, main storage, and the processor. These resources, in addition to program activations and invocations allow the units of work to be done.

The attributes of a process provide information relative to the limitations and to the importance of that process when using the machine resources. For example, these attributes limit the amount of processor resource a process can consume, limit the amount of temporary auxiliary storage it may consume during its existence, and also limit its access to objects (authorization).

While processes may coexist as independent entities in the machine, they may also require synchronization and communication with other processes that concurrently exist in the machine. These facilities are provided through queue and event management functions.

ESTABLISHING A PROCESS

A process is established by the Initiate Process instruction. The operands of this instruction specify the process control space and the process definition template, which are the fundamental elements of the process structure.

Process Control Space

The PCS (process control space) is a system object that must be supplied as a machine work area to support the execution of a process. The system pointer to the process control space is used as the target operand when referencing a process in process control instructions and also when signaling process-directed events to a process.

The PCS can be used by only one existing process at a time and cannot be destroyed while it is associated with a process. A PCS is associated with a process when it has been specified as the process control space operand on an Initiate Process instruction. When additional work area is required during the existence of a process, the machine implicitly extends the PCS.

Process Definition Template

The PDT (process definition template) specifies the set of attributes that establishes the framework for control of the process. These attributes are made up of both pointers and scalars. The pointers address objects that are used by the process for authorization verification, program work areas, programs to be invoked, and address resolution. The scalar attributes represent resource limits, process control information, and process characteristics.

Process Structure

Two key constructs are required for process initiation. One, the process control space, has been previously discussed. The second, the PASA (process automatic storage area), is specified as an attribute in the PDT and is represented by a space pointer that addresses the base entry. The PASA is used to allocate space for program objects with the automatic attribute.

Another construct, the PSSA (process static storage area), is also specified in the PDT, and is required prior to activating a program within the process that has static program objects. The PSSA is represented by a space pointer that addresses the base entry.

Process Initiation Steps

The initiation of a process occurs in two steps, synchronous and asynchronous. The synchronous initiation step is executed as part of the process that is causing the new process to be initiated. In addition, the synchronous initiation step uses the resource attributes currently in effect for the initiating process. In this step, certain key attributes are verified. Problems detected during the step are communicated to the initiating process as exceptions.

The asynchronous initiation step is executed as an independent unit of work and operates under the resource attributes for the new process. During this step, the new unit of work (process) attributes are verified. Any problems detected during this step are communicated via events. At the successful completion of both initiation steps, the process is said to exist and can then be the target of other process management functions and process-directed event signals.

Process Domain

When initiated, a process is defined to be either dependent on or independent of the existence of its initiator. Once established, the dependent/independent attribute cannot be altered for a process.

The domain of a process is defined as the set of processes that include the process itself and all dependent processes initiated by the process or by any subordinate dependent processes. In addition to controlling the existence of a process, the domain concept is used to determine the extent of the process suspend and resume operations.

PROCESS STATES

After being successfully initiated, a process enters an external existence state called the active state. On the initial entry of the process into this state, the machine provides the control for sequencing the process through three internal processing phases: initiation, problem, and termination. These phases establish a problem solving environment, solve the problem, and then complete the problem solving activity.

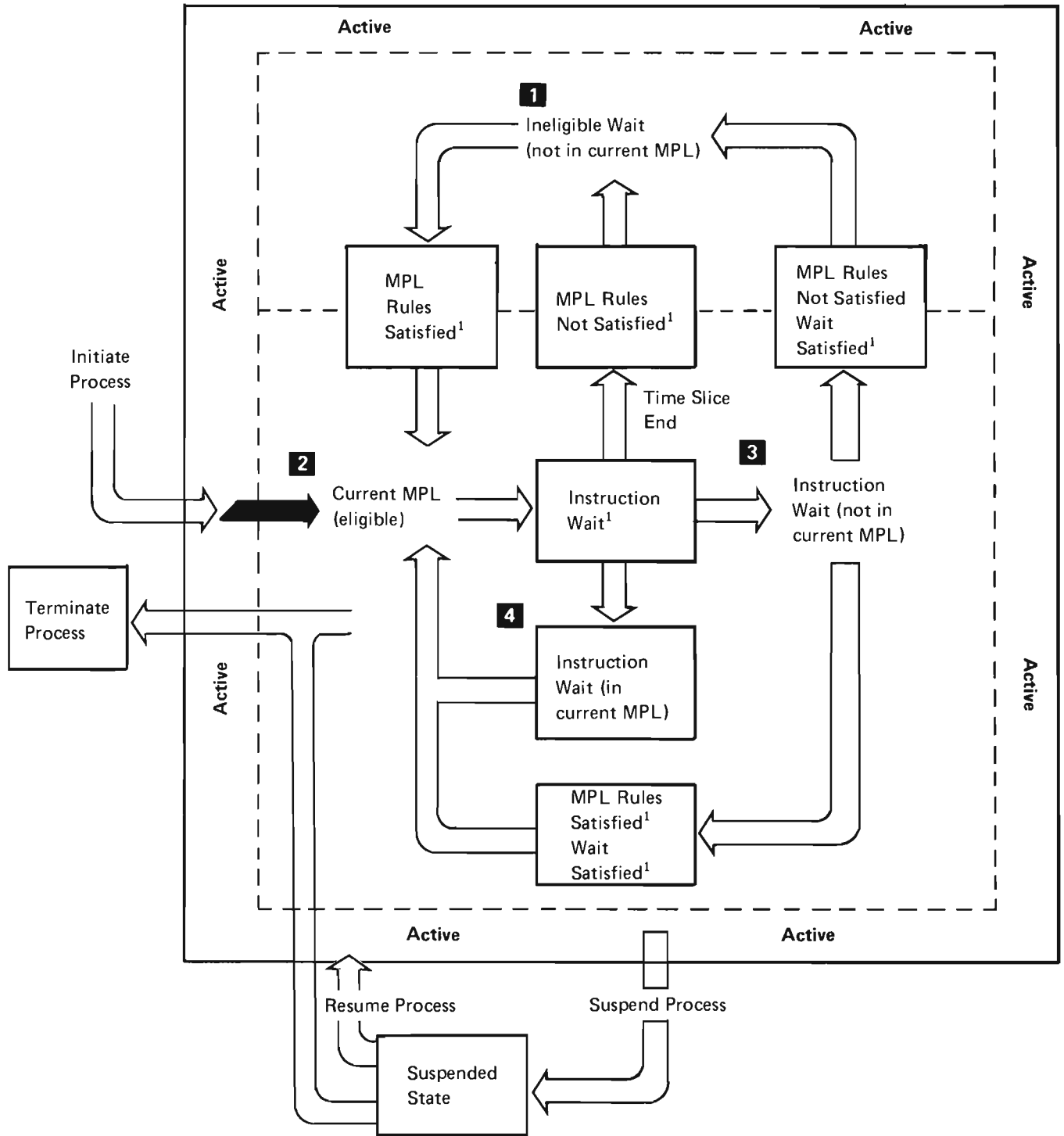
Upon completion of the appointed work or in any one of the internal processing phases, a process can be either terminated (in which case the process is purged from the machine) or placed in the external existence state (called the suspended state). In the suspended state, a process retains control of the designated resources that were acquired while in the active state. A suspended process can either reenter the active state and be sequenced through the remaining internal processing phases or be terminated. A process in the suspended state consumes none of the processor resources and consumes only enough main storage to maintain its existence as a process.

A process in the active state is defined to be at all times in one of four substates: ineligible wait, current multiprogramming level (MPL), instruction wait, or instruction wait in current MPL (see Figure 4-1).

Figure 4-1 is a state transition diagram that shows the external existence states and the active substates of a process. The required actions for the transitions to take effect are also shown.

- 1 Ineligible wait (not in current MPL)—A process is in this substate when it is neither executing nor in an instruction wait substate. The process is placed in this substate as a result of applying MPL rules.
- 2 Current MPL (eligible)—A process is in this substate when it is currently executing; that is, the process is externally viewed as consuming the processor resource.
- 3 Instruction wait (not in current MPL)—A process is in this substate if it is suspended or waiting for a resource, an event, a queue message, or an update access to a data space entry, and user direction has caused it to be removed from the current MPL.
- 4 Instruction wait (in current MPL)—A process is in this substate if it is waiting for the arrival of a message on a queue and if the user has explicitly directed the machine to make exception to the normal MPL rules by allowing the process to remain in the MPL during the wait interval.

For more information about the active substates of a process, refer to *Resource Management Attributes* later in this chapter.



¹ Action(s) required for the transition to occur.

Figure 4-1. State Transition Diagram

PROCESS PHASES

The internal processing phases (initiation, problem, and termination) are presented to the machine in the form of programs to be invoked as the sequencing proceeds. These programs are specified as process attributes contained in the process definition template.

The three process phases support the following major objectives:

- Minimize the required program modification so that a program can either be invoked directly within the process for synchronous execution, or be invoked as the root program in a new process for asynchronous execution.
- Allow for monitor type program functions to be performed by the process without having a monitor program invocation as the first program invocation in the problem phase.
- In all normal and abnormal conditions that can cause a process to be terminated, ensure that control can be acquired by supervisory programs executing within the process to complete the required termination functions before the process is destroyed.

The problem phase is the basic process phase in which problem solving work is done. The first program invoked in the problem phase has access to arguments passed to the process from the initiator of the process. The argument/parameter capability is functionally identical to the capability available between invocations within a process.

The initiation, problem, and termination phases are optional in terms of their use during the execution cycle of a process. In all cases, a process can enter either the initiation phase or the problem phase. Entry into the initiation phase is determined by the setting of the process attributes at the time the process is initiated. Within the initiation phase, there is no restriction in terms of the functions that can be performed. The only functional difference between the initiation and problem phases is that during the initiation phase, external arguments are not presented to the first invocation. In all cases, the user must specify either the initiation phase or the problem phase attribute. If needed, however, both the initiation and the problem phase attributes can be specified.

Work completed during the initiation phase can, in three basic ways, influence subsequent processing in the problem phase. First, the process attributes may be altered, including the specification of the first program to be invoked in the problem phase. Second, functions performed in the initiation phase have a process-wide persistence; that is, system objects created, locks obtained, and event monitors established are in effect when the problem phase is entered. Finally, programs executed in the initiation phase can force a process to the termination phase without entry into the problem phase.

Process management allows a process to enter the termination phase at the completion of normal execution in either the initiation or the problem phase based on an explicit instruction or as a result of any abnormal situation within the process. Additionally, in most situations in which a process termination is forced by an external action, the process can enter the termination phase. The external action referred to is a Terminate Process instruction issued from one process to another process.




Sequencing through Process Phases

As discussed previously, a process in the active state can be sequenced through three internal processing phases: initiation, problem, and termination.

During the initiation of the process, the machine determines whether the current process attributes specify the execution of a program in the initiation phase. If so, the program designated by the current process attributes is invoked. The process executes in this phase until either a Return instruction is issued from the highest level invocation or a situation or instruction is encountered that forces process termination.

After completion of execution in the initiation phase or if the process attributes specify no execution of programs in the initiation phase, the process automatically sequences to the problem phase. If the problem phase program is specified, the program designated by the current process attributes is invoked. Again, the process executes in the problem phase until either a Return instruction is issued from the highest level invocation or a situation or instruction is encountered that forces process termination.



After completion of execution in the problem phase or when any situation or instruction forces process termination, the process automatically sequences to the termination phase. If the current process attributes specify entry into the termination phase, the designated program is invoked. The process executes in the termination phase until either a Return instruction is issued from the highest level invocation or a situation or instruction is encountered that forces process destruction. In either of the preceding cases or if no programs are executed in the termination phase, the process is terminated.

PROCESS AUTHORIZATION

When a process is initiated, the process definition template is provided by the Initiate Process instruction. A system pointer, which is contained in the template, addresses a user profile that is used to govern the execution of the process. The user profile that currently governs the initiating process must have object management authorization for the user profile of the initiated process.

The process user profile provides the basic authorization control for the process. Also, permanent system object storage allocation and ownership of objects created by the process are reflected in the process user profile currently active at the time that a permanent system object is created.

While the process is in either the active or suspended state, a new process user profile can be specified for a process with the Modify Process Attributes instruction. In addition to the required process control eligibility required to specify the instruction, the user profile currently governing the process that issues the Modify Process Attributes instruction must have object management authorization for the new user profile.

A system pointer to the current process user profile is made available through the Materialize Process Attributes instruction. Any authorization set in the system pointer when it was last specified is returned when the process user profile pointer attribute is materialized.

The authorization for a process can be augmented through the invocation of programs created with the adopt user profile attribute. Adopted user profiles are used in conjunction with the process user profile to determine the eligibility of a process to access existing objects, privileged instructions, or special authorizations. When authorization verification is required, one or more currently adopted user profiles are used in combination with the current process user profile to determine the eligibility of a process for a restricted object or operation. These currently adopted user profiles augment the authorization of a process so long as the adopting program is currently invoked within the process.

OBJECT ADDRESS RESOLUTION

System and data pointers can be resolved within the process structure. System pointers can be resolved through the facilities of the NRL (name resolution list). Data pointers are resolved through program activations within the process.

The NRL is a process attribute specified in the process definition template. The NRL contains a count of the number of entries followed by the list of entries. The entries are resolved system pointers addressing contexts.

The NRL establishes both the identity and the sequence in which contexts are searched in attempting to resolve a system pointer to a system object. The NRL controls the system object address resolution when a specific context is not designated either in a system pointer with an initial value or in the Resolve Addressability instruction context operand.

The contexts in the NRL are searched in their order of appearance until an addressability entry for the proper object type, subtype, and object name is located, or until the list is exhausted.

The entries in the list can be modified at any time. In addition, an entirely new NRL can be specified by modifying the process attribute with the Modify Process Attributes instruction.

External Data Object Resolution

A data pointer defined in a program that is invoked within a process can be resolved to assume the address and attributes of a symbolically identified variable. The subject variable must have been defined with the external attribute and must reside in an activation entry that exists in the process when the resolution is attempted.

During data pointer resolution, if a program is specified in addition to the variable name, then only the activation associated with the identified program is searched for the specified variable. If no program is specified, then all programs associated with activation entries in the process are searched, starting with the most recent activation and continuing to the oldest. When no specific program is specified (or if the program specified is the current activation), the activation that contains the instruction causing address resolution also is included in the search. The activation entries are contained in the PSSA (process static storage area).

PROCESS MANAGEMENT INSTRUCTIONS

There are two categories of process management instructions. One category provides capabilities for creating and destroying the process control space, which is the basic element of the process structure. The second category provides functions for controlling the existence and states of processes (process control).

Process Control Space Instructions

The Create Process Control Space instruction creates the work area required by the machine to support program execution within a process. The size of the work area is determined by the machine and, if required, is implicitly extended by the machine. Once a process is associated with the PCS (process control space), the system pointer that addresses the PCS can be used as the process identifier on all process control instructions.

The Destroy Process Control Space instruction destroys the specified PCS. A process control space cannot be destroyed while it is associated with a process; that is, the PCS has been specified for a currently existing process.

Process Control Instructions

The Initiate Process instruction causes a new process to be established within the specified process control space. The attributes for the process are specified in the PDT (process definition template). The system pointer that addresses the PCS is then used for process identification on the Terminate Process, Terminate Instruction, Suspend Process, Wait on Time, Resume Process, Materialize Process Attributes, and Modify Process Attributes instructions.

The Terminate Process instruction causes the internal processing phase of a process to be altered to the termination phase. The termination phase program is invoked if the process is not already in that phase. If the process is already in the termination phase, this instruction can cause the process to be terminated, or the process can optionally be left in the termination phase.

Once a process no longer exists, the process control space formerly associated with the process can be specified for the initiation of a new process.

The Terminate Instruction instruction causes certain long-running MI instructions in a process to terminate. This instruction tries to terminate an instruction that is currently executing.

Only the following instructions, which require a relatively long execution time, are subject to termination:

- Activate Cursor (ACTCR)
- Apply Journalled Changes (APYJCHG)
- Copy Data Space Entries (CPYDSE)
- Create Data Space Index (CRTDSINX)
- Create Program (CRTPG)
- Data Base Maintenance (DBMAINT)
- Insert Sequential Data Space Entries (INSSDSE)
- Modify Data Space Index Attributes (MODDSIA)
- Request I/O (REQIO)
- Retrieve Journal Entries (RETJENT)
- Retrieve Sequential Data Space Entries (RETS DSE)
- Set Cursor (SETCR)
- Signal Exception (SIGEXCP)

The Signal Exception instruction can also signal the terminate exception, but it is not actually subject to termination because it is not a long-running instruction.

If one of these instructions is not the current instruction executing, the termination request is ignored. When an instruction is terminated, it signals an exception.

The Suspend Process instruction causes a process to be placed in the suspended state. In this state, the process is not eligible to compete for the processor resource. A suspended process can be terminated or modified; it can also hold locks, receive transferred locks, and receive event signals.

The Wait On Time instruction causes the process to wait for a specified time interval.

The Resume Process instruction causes a process to be placed in the active state. In this state, the process can compete for the processor resource under the normal MPL (multiprogramming level) rules.

The Materialize Process Attributes instruction causes materialization of the current attributes of the process. In addition to the attributes supplied on process initiation or modification, there are resource usage, subprocess identification, and process performance attributes that can be materialized. Some examples of this information are:

- Total processor time used by the process
- Total temporary auxiliary storage used by the process
- Number of locks held by the process
- List of process control space system pointers for immediately subordinate processes
- Number of transitions into ineligible wait state
- Number of transitions into instruction wait
- Number of transitions into ineligible wait from instruction wait
- Time stamp upon entering last ineligible wait state

The Modify Process Attributes instruction causes one of the defined set of process attributes to be modified. Depending on the attribute, the modification may manifest itself immediately, or there may be a time delay before the attribute is referenced by the machine.

Authority for Process Control Instruction Usage

Two categories of authority allow the use of process control instructions. First, there is an implied authority granted to the initiator of a process that allows the initiator to terminate, suspend, resume, modify, or materialize an immediately subordinate process without restriction. This implied authority is also granted to a process taking action against itself, excluding process attribute modification that places restrictions on when internal modification can be allowed.

In all other cases, the process issuing the process control instruction must have process control special authorization in either the most current adopted user profile or the process user profile.

Additionally, a process issuing the Initiate Process instruction must have privileged instruction authorization for the instruction, or an exception is signaled.

PROCESS ATTRIBUTES

Process attributes refer to the required information for the machine to establish the framework for the control of a process. The attributes are established via the process definition template (PDT) when a process is initiated. Most process attributes are eligible for modification (with certain limitations) either by the process itself or by other processes.

The following information describes the significance of each process attribute and the associated machine functions.

Process Control Attributes

The process control attributes control those actions taken by the machine for certain temporary conditions encountered by the process. Some of these attributes establish the relevance of other process attributes.

The process type attribute specifies whether the existence of the new process is to be dependent on or independent of the existence of its initiator.

The instruction wait access state control attribute provides a process level capability to disallow the modification of the access state of the process's access group upon entry to and exit from an instruction wait even though the instruction specifies access state modification.

The time slice end access state control attribute allows specification of whether or not the access state of the access group for the process is to be modified at the end of a time slice for the process.

The time slice event option attribute specifies whether the event is to be signaled when a process has exhausted its time slice without having entered an instruction wait.

The initiation phase option attribute specifies whether the initiation phase program attribute is supplied and, consequently, whether the initiation phase is to be entered.

The problem phase option attribute specifies whether the problem phase program attribute is supplied and, consequently, whether the problem phase is to be entered.

The termination phase option attribute specifies whether the termination phase program attribute is supplied and, consequently, whether the termination phase is to be entered.

The process default exception handler option attribute specifies whether a process default exception handling program is supplied. If supplied, this program is invoked when an exception occurs that is not handled by the invocation to which the exception was signaled.

The process name resolution list option attribute specifies whether the name resolution list pointer attribute is supplied and, consequently, can be used by the machine for object address resolution through the context specifications.

The process static storage option attribute specifies whether the PSSA pointer attribute is supplied. A PSSA pointer must be supplied prior to activating or invoking the first program with static storage requirements.

The process access group option attribute specifies whether the process access group pointer attribute is supplied, and can therefore be referenced by the machine during access state modification as a result of the process leaving or entering the MPL.

The signal event control mask attribute allows the signaling of events to be conditioned at a process level. By appropriately setting the mask, the process can preclude the conditional event signals coming from within the process while unconditional signals are allowed to occur.

The number of event monitors attribute allows the machine to more effectively manage event monitors. This performance variable is not a maximum value; however, when this performance variable is used in relation to the event monitors signaled most frequently within the process, it achieves a positive performance impact.

Resource Management Attributes

The resource management attributes define resource consumption limitations, relative importance of a process in the presence of other processes, and direct the machine to place the process in the proper subdivision of processes contending for like resources.

The process priority attribute establishes the importance of one process in relation to all other processes in the machine. At the most fundamental level, the priority of a process establishes its position in the order of competing processes.

The process storage pool identification attribute defines the identification of the main storage pool from which the main storage demands for the process are satisfied.

The maximum temporary auxiliary storage allowed attribute places a limit on the amount of auxiliary storage consumed during the existence of a process relative to the creation and extension of temporary objects. An event is signaled when this limit is exceeded.

The time slice interval attribute specifies the maximum amount of processor time a process can consume before involuntarily relinquishing control of the processor resource.

The default wait time-out interval attribute specifies a real time interval representing the maximum amount of time a process will wait for an object lock, the arrival of a message on a queue, or an event to be signaled in the absence of a wait interval specified on the applicable instruction. An exception is signaled if the interval expires prior to satisfying the conditions causing the wait.

The maximum processor time allowed attribute limits the amount of processor time a process can consume throughout its existence. A machine event is signaled when this limit is exceeded.

The process multiprogramming level class ID attribute specifies the identification of the MPL class to which the process is to be associated. MPL classes allow the set of existing processes to be divided into subsets. The elements of the subset then compete for the machine resources only with other elements of the subset. For more information about MPL classes, refer to *Resource Management* later in this chapter.

The modification control indicators serve to control the modification of the attributes of a process either by the process itself or by an external process. The modification can be allowed or disallowed in internal processing phases based on the indicator settings. A set of indicators exists for each modifiable process attribute.

Process Pointer Attributes

The process pointer attributes are a set of system and space pointers that address the various objects required for the execution of a process; for example, addressability to programs, a user profile, and process storage areas.

Process User Profile: The user profile provides the authorizations for privileged instruction usage and object access. It is also used to limit and record the amount of auxiliary storage used for the creation of permanent objects for all processes using the user profile.

Process Communication Object (PCO): This attribute is a user convention and is not validated by the machine. The PCO can be used to pass information from one process to another outside the conventional interfaces such as queues or events. If the PCO is a space pointer, programs executed in the process can have variables declared based on this pointer.

Process Name Resolution List (NRL): The NRL is a list of contexts used for system object address resolution when a context is not specified on initial-valued system pointers or resolve addressability operands.

Initiation Phase Program: When specified, this program is the first program invoked when the process enters the initiation phase.

Termination Phase Program: When specified, this program is invoked when the process enters the termination phase due to: an exception not being handled by the process, a Return External instruction being issued by the problem phase program, or a Terminate Process instruction being issued against the process.

Problem Phase Program: When specified, this program is invoked when the process enters the problem phase; either the initiation phase program or the problem phase program must be specified.

Process Default Exception Handler: When specified, this program is invoked when the signaled invocation does not handle the exception.

Process Automatic Storage Area: This space is used for automatic program object allocation upon invocation of a program.

Process Static Storage Area: This optional space is used for static program object allocation during program activation. It must be specified prior to activation of a program requiring static storage.

Process Access Group: This attribute designates an access group that is transferred to and from auxiliary storage as the process exits and enters the current MPL. The access state modification (transfer) is conditioned by the following options: instruction wait access state control indicator, time slice end access state control indicator, and the instruction wait access state modification. For more information about access groups, refer to *Resource Management* later in this chapter.

Process Status Indicators

The process status indicators reflect information relating to the external existence state, internal processing phase and, possibly, termination status. More specifically, the indicators denote the following types of information that relate to the current status of a process:

- Active or suspended. If the process is active, its active substate is: ineligible wait, instruction wait, or currently in MPL.
- Internal processing phase (initiation, problem, or termination).
- Process interrupt pending status (for example, terminate process pending, suspend process pending, or transfer lock pending).
- Process termination status. The reason for termination of the process (return from first invocation in problem phase, exception not handled, or terminate process affected) and the termination code.

This information is returned via a Materialize Process Attributes instruction option.

Process Resource Usage Attributes

The following attributes reflect the current consumption of resources by a process. These attributes are returned via a Materialize Process Attributes instruction.

The total temporary auxiliary storage used attribute specifies the amount of temporary auxiliary storage currently allocated to a process as a result of temporary object creation and extension during the existence of a process.

The total processor time used value represents the current consumption of the processor resource by a process.

The number of locks currently held by the process value denotes the number of locks held on system objects. This value includes implicit locks acquired by the machine.

Subordinate Processes Identification

This attribute is represented by a list of system pointers that address process control spaces associated with immediately subordinate processes. Preceding the list is a count of the number of immediately subordinate processes. This attribute is available via a Materialize Process Attributes instruction.

Process Performance Attributes

These attributes provide an indication of how the process is proceeding by showing the number of times the process has voluntarily relinquished its place in the current MPL and the number of times the process was displaced in contention for a place in the current MPL. These attributes are returned via a Materialize Process Attributes instruction.

INTERPROCESS COMMUNICATION


Interprocess communication can be effected at process initiation and dynamically during process execution.

The two vehicles for communicating information to a process during initiation time are the PCO (process communication object) and the argument/parameter list interface. The PCO is specified as a process attribute in the PDT (process definition template), and its attribute is a user convention. The new process accesses the PCO through use of the Materialize Process Attributes instruction.

The argument/parameter interface provides the same function as that for program invocation. The argument list is specified as an operand on the Initiate Process instruction and is only mapped to the problem phase program. This function allows the program represented by the problem phase program to be invoked as an asynchronous or synchronous unit of work without requiring the program to have knowledge of this characteristic.

A more dynamic means for interprocess communication, and also affording synchronization capabilities, are those functions provided by queue and event management.

A queue is an object that provides the capability for a process to wait until the required data becomes available. In addition, the function to synchronously test for the availability of the required data is also provided. The required data is typically a message enqueued to the queue by another process. Some messages, however, come from the machine as in the case of a message generated in response to a device support request (I/O operation). The process requiring the data attempts to dequeue a message from the appropriate queue and when the message is available, the process can be made eligible to enter the MPL. For more information about queues, refer to *Queue Management* in Chapter 5.



Event management provides a similar but expanded function; the data is an event signal. Like queues, the process can both wait and test for the occurrence of the event signal. Two additional features are provided by events: the process interruption and the broadcast event signals.

The process interruption facility allows the temporary suspension of the normal execution sequence of a process. A process can be interrupted at an instruction boundary, in an instruction wait (Lock, Dequeue, and Wait on Event), or in a long-running interruptible instruction when an event signal occurs. When the event occurs, a program (event handler) can be invoked to handle the event signal. When the event handler function is completed, process execution is resumed at the point of interruption.

The broadcast capability allows an event signal to be received by multiple processes; each process can then take action appropriate to its function.

OBJECT LOCKS

Objects are locked at the process level; that is, a process is considered to be the holder of a lock. In addition to the dynamic acquisition of locks afforded by resource management instructions, locks can be transferred to a new process during the time the process is initiated. The locks to be transferred are represented by an operand of the Initiate Process instruction. Rules for lock transfers that apply to the Transfer Lock instruction apply also during process initiation; for example, the initiator of the process must be the holder of the lock. Lock states that cause conflicts cannot be transferred.

An implicit lock is acquired during process initiation for the user profile of a new process.

For more information about locks, refer to *Resource Management* later in this chapter.

PROCESS EXCEPTION HANDLING

Exceptions are either machine-defined errors detected during instruction execution or user-defined conditions detected by user programs. Generally, exceptions are error conditions, but they can be a means for communicating information to other invocations that currently exist within the process.

Exception description program objects specify the action to be taken when an exception is signaled or resignaled to a program invocation; exceptions are signaled or resignaled only to a specific invocation. If the exception is not handled by the signaled invocation, control is given, if specified, to the PDEH (process default exception handler).

The PDEH, as mentioned in the explanation of process attributes, is identified by a system pointer addressing a program in the PDT (process definition template). The PDEH is given control under the following conditions:

- The signaled invocation does not handle the exception (refer to *Exception Management* later in this chapter for the concepts of exception handling).
- An exception is signaled or resignaled from the only currently existing invocation in the process.

The PDEH is invoked as an external program following the most recent program invocation, and the Return from Exception instruction must be issued to exit from the PDEH.

For more information about exception-related functions, refer to *Exception Management* later in this chapter.

PROCESS CONTROL INSTRUCTION CHARACTERISTICS

The concept of a synchronous/asynchronous step characteristic for the initiation of a process is described under *Process Initiation Steps* earlier in this chapter. That is, a portion of the initiation function is accomplished under the resources of the initiator, and the remainder is performed under the new unit of work (process). The following process-related instructions also have this characteristic:

- Terminate Instruction
- Terminate Process
- Suspend Process
- Resume Process
- Modify Process Attributes

This means that when control is returned to the next sequential instruction following one of the previous instructions, the function has already been scheduled. The point at which the function will be completed is dictated by resource management constraints such as priority, MPL rules, and main storage availability.

Event Management

Event management functions provide the user with the capability to monitor the occurrence of a set of events and take action based on the occurrence of some or all of that set of events.

EVENTS

Events relate to and define activities that occur during machine operation that may be of interest to users of the machine. Event management enables a process to monitor these events and perform specific functions based on their occurrence. The events being monitored can represent conditions either internal or external to the monitoring process; that is, one process can be monitoring conditions caused by the same process, by other processes that are currently in the machine, or by some condition not directly related to the existence of any process in the machine.

Machine events are signaled as a result of certain conditions detected by the machine. Explicitly signaled events can be defined (based on user protocol), signaled, and monitored through the event management facilities. Events can be signaled to event monitors in all processes or to a specific process. Generally, a process monitors an event so that some function, which is related to the conditions that caused the event, can be performed.

There are two major classifications of machine events: object-related and machine-related.

Object-related events are conditions directly related to the values and the attributes of an object. For example, the message limit is reached on a queue, or a network description line failure occurs.

Machine-related events are conditions that are not directly related to a system object. For example, a specific timer interval has elapsed, or a machine error has occurred.

For a complete list of machine events, refer to *Event Specifications* in the *System/38 Functional Reference Manual*.

In addition, there is a group of events that have no meaning to the machine because they are defined outside the machine range. These events are based on user protocol. The same event management facilities available for machine events are also available for explicitly signaled events and, therefore, may be used for intraprocess monitoring and interprocess communication and synchronization.

Event Identification

Events are identified by the following components.

- **Event class**—The class of events is generally related to a specific type of object, a machine function, or a condition. For example:
 - Queue events
 - Authorization events
 - Machine status events
- **Event type**—Type is the subclassification of the event within a class to further identify the event. For example, subtypes are defined for the class of machine status events, the machine check, machine power, and error log full.
- **Event subtype**—Subtype is a further subclassification of the event within the type. For example, within the machine power type, the subtype specifies the machine power status such as a power controller failure.

A specific event can be monitored by specifying the entire event identification (class, type, and subtype). Events can also be monitored generically. Generic monitoring can be by class (the occurrence of any event within the class without regard to type or subtype) or by type (the occurrence of any event within a particular class and type without regard to subtype).

Event monitoring can be further limited by specifying a qualifier. The qualifier is termed a compare value and can be a reference to a system object (through a system pointer) or a scalar.

Events related to system objects can be monitored for all occurrences of an event (for example, the message limit reached on any queue). Events can also be qualified (by specifying the appropriate event monitor compare value) such that monitoring occurs only if the event occurs for a specific system object (for example, the message limit is reached on queue XYZ).

Events that have associated compare values can be monitored so that their occurrence is detected only if a specified compare value matches the condition (compare value) in the event. For example, timer events are signaled if the time interval specified in a compare value has elapsed or if the time of day specified in a compare value has been reached.

EVENT MONITORING

The occurrence of an event can be monitored by establishing an event monitor that describes which conditions are to be monitored and how the event is to be handled when it occurs. The event monitor is explicitly established by the Monitor Event instruction. The event monitor can be terminated explicitly by the Cancel Event Monitor instruction or implicitly by process termination.

The specific attributes and values of an event monitor are as follows.

Event Identification: Events are identified by the following components:

- Event class—(For example, queue events, process events, LUD events.) The event class is represented by a value from hex 0001 through hex 7FFF.
- Event type—Type of event within a class to further identify the event (for example, create, modify, or destroy). The event type is represented by a value from hex 00 through hex FF.

Type 00 is never signaled; this value is restricted to support the technique of generic monitoring for any event type within a class.


- Event subtype—The various differences within each type to further identify the event. The event subtype is represented by a value from hex 00 through hex FF.

Subtype 00 is never signaled; this value is restricted to support the technique of generic monitoring for any event subtype within an event type. If a generic event type is specified (value of zero), the event subtype must also be generic (value of zero).

Compare Value Qualifier: There are certain classes of machine events that allow or require a compare value to be specified when an event monitor is established. This compare value further qualifies the event monitor.

For process events, the compare value represents the process identification that is used to monitor the change in the status of a specific process. The compare value can contain a system pointer, but the system pointer must be located in the first 16 bytes of the compare value.

For timer events, the compare value specifies the time-of-day or real-time interval and optional user data to cause the event monitor to be signaled.




Event Monitor Priority: Specifies the importance of this event in relation to other events being monitored by the process. When multiple events occur, this value establishes the order that event handlers will be scheduled; the event with the highest priority is scheduled first. In addition, when a process is waiting for a specific event monitor, this value determines whether that process can handle a different event monitor.

Enabled/Disabled State: Specifies whether the event monitor is enabled for receiving. The state can be altered by the Enable Event Monitor and Disable Event Monitor instructions.

Signal Retention Option: Specifies whether signals are to be retained while the event monitor is disabled. When specified, this option overrides the maximum number of signals to be retained value.

Short Form Option: Allows the event monitor to specify that only part of the event-related data is to accompany the signal.



Note: An improvement in performance might be realized if this option is specified.

Maximum Number of Signals to be Retained: Defines the number of signals and associated event-related data to be retained while the process is masked, the event monitor is disabled, or the event monitor is enabled with the events not being handled as rapidly as they are signaled. This number and the signal retention option, not the disabled/masked states of the event monitor or process, are the controlling factors relative to the machine discarding signals.

Event Handler Specification: Identifies the event handling routine to be given control when an event occurs. The event handler specification is a system pointer that addresses a program or the activation of a program. The event handler specification is optional, but it must be defined for the event monitor if the associated event is to be handled asynchronously.

Monitor Domain: This attribute defines whether the event monitor is to be informed of events that are signaled machine-wide or only those events that are directed to the process with which the event monitor is associated. If the monitor domain is specified as process directed, it will be signaled only if the event is signaled directly to the process that contains the event monitor.

EVENT SIGNALING

The event signaling portion of event management defines the occurrence of an event, and then communicates that occurrence to a process through an event monitor. (All qualifying monitors are made aware of the event.) Any process that is monitoring for the occurrence of the event is notified through the event signaling mechanism.

Events can be signaled implicitly by the machine or explicitly by the Signal Event instruction.

Machine Event Signaling

The occurrence of machine events is made known either to a specific process or to any process monitoring for the event. The event monitor must correspond to the event being signaled. This correspondence is as follows:

- The event identification must match or specify a generic type or subtype that contains the signaled event.
- The compare value qualifier (if present) must match, for the length specified in the event monitor, the value in the event being signaled.
- The event monitor must define a machine event.

When an event is signaled to an event monitor, the data associated with the occurrence of the event is collected by the machine and saved for presentation to an event handling routine.

Signaling by Signal Event Instruction

Events can be defined to represent any specific condition established by a protocol. These events (both user-defined and machine) can be signaled to any event monitor in any process, or they can be signaled to a specific process by the Signal Event instruction. The Signal Event instruction specifies an event identification, a conditional signal mask, a signal domain, a compare value, and event-related data.

A nonzero value in the conditional signal mask causes the associated event to be conditionally signaled based on a comparison of the conditional signal mask and the signal event control mask process attribute. The signal event control mask can be modified from inside or outside the process in order to control the signaling of events. This capability allows groups of events to be selectively signaled on a process level basis.

An event with a machine-wide signal domain indicator is made available to all machine-wide event monitors in any process (including the signaling process) that corresponds as follows:

- The event identification must match or specify a generic type or subtype that contains the signaled event.
- The compare value (if present) must match (for the length specified in the event monitor) the value in the event being signaled.

If the event is signaled to any event monitor, the event-related data specified for the instruction is saved for presentation to an event handling routine.

Signaling an Event to a Process

Certain machine events are signaled directly to a process. User-defined events can also be signaled directly to a process. The Signal Event instruction can be used to confine a machine event to a specific process even though the event monitor is assigned a machine-wide domain. The Signal Event instruction with a process as the signal domain causes an event monitor in the specified process to be placed in the signaled state in the same manner as if the machine had signaled the event.

Event-related data is provided with the instruction and is saved by the machine for presentation to the event handling routine.

Conditions for Signaling an Event Monitor

When an event occurs (coming from the machine or based on an explicitly signaled event), the machine locates all event monitors in the processes qualified to receive the signal for that event. The following conditions must exist to activate a monitor, invoke a handler routine, or add to the pending signals for a monitor:

- The event identification, the compare value, and the event type must correspond as previously described.
- If the process is not masked and the event monitor is enabled, the event monitor is signaled.
- If the event monitor is disabled, the signal retention attribute of the event monitor determines whether signaling of the event monitor is to occur. If the signal retention attribute option is set to retain signals while the event monitor is disabled, the event monitor retains the signal until either the event monitor is enabled or a Test Event instruction is issued.
- If it is determined that the event monitor is to be signaled based on the previous conditions but the number of signals pending value of the event monitor is currently equal to the maximum number of signals to be retained, the event is not signaled to the event monitor. Otherwise, the event monitor is signaled, the number of signals pending value is incremented by one, and the event-related data is saved for later retrieval by the event handling routine.

EVENT HANDLING

Event handling refers to the portion of event management functions that relate to the scheduling of asynchronous invocations to the signaled process or to the synchronous testing for signaled events.

Asynchronous Event Handling

Asynchronous event handling refers to the invocation of a program to handle an event. This invocation occurs during program execution at a point dependent on the occurrence of one or more events rather than during the execution of a particular instruction.

The following conditions must be met before an event handling program is invoked:

- The event monitor must have defined an event handling program.
- The event monitor must have at least one signal pending.
- The event monitor must be enabled.
- The process must not be masked.
- There cannot be any enabled event monitors in the process with event handling programs of a higher priority and in the signaled state. There may be, however, higher priority but disabled monitors that have been signaled.
- The process cannot be waiting for an event monitor of higher priority to be signaled.

When an event monitor is signaled and the previous conditions are met, instruction execution is interrupted, and the event handling program is invoked in the process. Based on the conditions necessary for an event handling program to be invoked, the invocation can occur if all of the following conditions exist:

- An event is signaled to an event monitor.
- The process is unmasked.
- An event monitor is enabled.

On entry to the event handling program, the process is placed in a masked state and no other events can interrupt the execution of the event handler. However, the event handler can choose to execute the Modify Process Event Mask instruction to allow other event handlers (including itself) to be invoked. The event handler can then be interrupted for other event handling.

The event handling invocation can execute the Retrieve Event Data instruction to determine the conditions that caused the signal. The instruction presents the data related to the oldest signaling of the event associated with the event monitor. Once event-related data is retrieved for a signaled event, it is discarded by the machine and the number of signals pending count is decremented by one. When the number of signals pending count reaches zero, the event monitor is no longer considered to be in the signaled state.

If the associated event monitor has multiple signals pending, the Test Event instruction can be executed in an event handler as many times as there are signals, in order to retrieve multiple sets of event-related data.

When the event handling sequence is complete, a Return External instruction must be executed to return control to the point following the instruction that was executed before the interruption for event handling. At this time, if the event-related data associated with the current signal has not been retrieved, the data is discarded and the number of signals pending is decremented. The process is implicitly unmasked when the Return External instruction is executed.

Synchronous Event Handling

Synchronous event handling refers to the explicit testing for the occurrence of an event at predefined locations during process execution. The occurrence of the event can be tested by the Test Event instruction or the Wait on Event instruction.

The event monitor can be in the signaled state if it was signaled and one of the following conditions exist:

- The process is masked.
- The event monitor is disabled.
- The event monitor has specified that no event handling program is to be invoked.
- The process is waiting for an event signal.

The Test Event instruction causes the current signal status of an event monitor to be tested. If the event monitor is in the signaled state, the event-related data retrieved from the event monitor is the data associated with the oldest event that was signaled. The event-related data for that event is discarded, and the number of signals pending count is decremented by one. When the count reaches zero, the event monitor is set to the not signaled state. A branch can be taken, or indicators can be set based on the signaled or not signaled state of the event monitor.

If the Test Event instruction does not reference a specific event monitor, the highest priority event currently in the signaled state is tested. If no event monitor is currently signaled, the not signaled resultant condition is used for branching or for setting an indicator.

The Wait on Event instruction allows for the signal status of an event monitor to be tested, and if the event monitor has not been signaled, the instruction waits for the signal.

If the event monitor is in the signaled state when the instruction is executed, event-related data is retrieved by the instruction. The number of signals pending count is decremented by one, and if the count becomes zero, the event monitor is set to the not signaled state.

If the event monitor is not in the signaled state, the Wait on Event instruction causes the process to wait for the signal to occur. If the event monitor is not signaled within a specified amount of time, an exception is signaled.

The Wait on Event instruction can wait for a signal to any event monitor or to one particular event monitor. If the wait is for any event monitor, a signal to any monitor without an event handler satisfies the wait. If the event monitor has an event handler that is enabled, the event handler is invoked. After control returns from the event handler, execution continues at the instruction following the Wait on Event instruction. If an event handler is invoked, no event-related data is available through the Wait on Event instruction.

When the wait is for any event, a signal to any event monitor in the process that does not have an event handler causes the wait to be satisfied.

When the Wait on Event instruction is executed while the process is masked, an exception is signaled.

The Wait on Event instruction cannot be satisfied by an event monitor in the disabled state. Disabled event monitors can, however, be signaled while the process is waiting for an enabled event to be signaled.

When the Wait on Event instruction is waiting for a specific event monitor to be signaled, the signaling of an event monitor with a higher priority than the waited for event monitor causes the event handler for the higher priority event monitor to be invoked (if one is specified). When the event handler is complete, the wait continues.

EVENT-RELATED DATA

Event-related data is that information relating to the conditions for which the event was signaled.

When an event is received by a process, the event-related data is saved for presentation to an event handling program at some later time. Once the event-related data is retrieved for a signaled event, the data is no longer available from the machine. Event-related data is made available by the following instructions.

- **Retrieve Event Data**—Event-related data is presented to an event handling invocation. The data is associated with the event that was signaled to the event monitor and, in turn, caused the event handler to be invoked.
- **Test Event**—Event-related data is presented through the instruction if the tested event monitor is currently in the signaled state. The event-related data is associated with the oldest event signaled to the event monitor.
- **Wait on Event**—Event-related data is presented through the instruction when the conditions causing the wait are satisfied. The data is associated with the event that was signaled to the event monitor that satisfied the wait.

Event-related data consists of standard and event-specific data. The standard data includes the following:

- The number of signals pending
- The time of the event signal
- The identification of the process causing the signal (if applicable)
- The event identification
- The compare value (if applicable)
- The origin of the signal (machine or user via the Signal Event instruction)

Event-specific data contains information unique to each event. For a description of the event-specific data, refer to the *System/38 Functional Reference Manual*.

The following standard event-related data is not available if the event monitor has specified the short form option:

- Number of signals pending
- Time of the event signal
- Identification of the process causing the signal (if applicable)

In addition, no event-specific data is available.

EVENT RULES

The interaction between synchronous and asynchronous event handling, the instructions that control event monitors, and the special characteristics of the timer result in rules for users of events. These rules are summarized as follows:

- Timer events cannot be signaled through the Signal Event instruction.
- When an invocation is in a wait on event state and the waited on event is canceled by an event handler, then the wait will never be satisfied. The same situation is true of a disabled event monitor if another event handler does not enable the event monitor.
- When an event monitor is canceled while in the signaled state, all signals are lost. Establishing another monitor for the same event will not restore the signals.
- When the timer interval is too small, all processing in the machine could be consumed in servicing the timer. Consequently, there is a restriction on the minimum value for the time interval.
- Monitor event and cancel event monitor sequences can appear anywhere within a process. Thus, it is possible to establish an event monitor within the initiation phase of a process to execute its event handler in the initiation, problem, and termination phases and to cancel the event monitor in the termination phase of the process.
- Event handlers are scheduled on the basis of the priority established through the Monitor Event instruction. During the time between the start of the schedule of an event handler and the actual invocation of the event handler, a higher priority event may be signaled to the process. Even so, the schedule of the lower priority event handler is *not* interrupted in deference to the higher priority event handler.

Exception Management

Exceptions are either errors detected by the machine as a result of executing an instruction or conditions detected by user programs. Exception management relates to the detection and signaling of exceptions by the machine and the monitoring for and handling of user-defined exceptions.

Two major classes of exceptions are defined: machine and user.

Machine-defined errors are detected during instruction execution. The errors that can be detected for a given instruction are contained in the *System/38 Functional Reference Manual*. The identification values reserved for machine exceptions range from hex 00 through hex 7F. However, a user is not prohibited from signaling a machine exception in order to simulate the occurrence of a machine exception.

User-defined conditions are detected by user programs. These conditions are defined outside the range of and have no meaning to the machine. Exception management facilities can be used to communicate, monitor, and handle these exceptions.

Exception management monitors both the occurrence of unexpected errors or conditions and the occurrence of expected errors or conditions that occur at unexpected times. A user can monitor any number of exceptions and, based on their occurrence, ignore the exception, note the occurrence of the exception, or attempt to recover from the exception. When the occurrence of an exception is not monitored, the machine takes a default action, which includes invoking a process default exception handler (defined as a process attribute), or terminating the process if no process default exception handler is defined.

EXCEPTION DESCRIPTIONS

The user monitors for the occurrence of an exception through an exception description.

An exception description is a program object and, therefore, is defined as an object in the ODT (object definition table). The exception description relates the execution of the program (or possibly a program invoked by the program) to the possible occurrence of an error. Exception descriptions monitor exceptions only when the defining program is invoked.

Exception descriptions define the set of exceptions to be monitored. An exception description can monitor for the occurrence of:

- All exceptions
- A class of exceptions
- A specific exception

Multiple exception IDs can be specified in an exception description to allow the monitoring of multiple classes, multiple exceptions, or both.

The exception description specifies an action to be performed based on the occurrence of one of the exceptions defined in the exception description. One of the following actions can be selected:

- Do not handle—Ignore the exception and continue instruction execution at the point where the exception occurred.
- Do not handle—Continue the search within the signaled invocation for another exception description monitoring the exception.
- Do not handle—Continue the search for another exception description monitoring the exception by resignaling the exception to the immediately preceding invocation.
- Defer handling of the exception by specifying that the machine is to record the occurrence of the exception so that its occurrence can lead to an action being performed at a later time. Execution of the instruction continues at the point where the exception occurred.
- Act immediately upon the exception by invoking another program in the process, invoking a subinvocation in the same program where the exception description is defined, or causing a branch to be executed to a designated instruction in the program where the exception description is defined.

If the exception is to be handled immediately, the exception description must define the exception handling mechanism by specifying one of the following options:

- Branch to a specific instruction when the exception occurs.
- Invoke an internal subinvocation when the exception occurs.
- Invoke a program when the exception occurs.

The exception description can further qualify the monitoring of exceptions by specifying a compare value (a maximum of 32 bytes). When the compare value is specified, the exception description monitors the exception only if the exception is signaled by the Signal Exception instruction, and a compare value is specified in the instruction which corresponds to the compare value in the exception description.

An exception description, which qualifies to handle an exception based on its exception ID, need not specify a compare value even though a compare value was specified in the Signal Exception instruction.

Each machine exception has an identification (hex values 0000 through 7FFF are reserved for machine exceptions). User-defined exceptions have identifications as specified by user conventions.

Once the defining program is invoked, the current attributes and values of the exception description can be materialized or modified. The Materialize Exception Description instruction can be used to materialize an exception description, and the Modify Exception Description instruction can be used to modify an exception description.

The following exception description attributes can be materialized or modified:

Attribute	Materialized	Modified
Exception handling action	Yes	Yes
Exception handler specification	Yes	No
Exception identification	Yes	No
Compare value	Yes	Yes

The exception handler specification defines whether the exception is handled by a branch, an internal subinvocation, or an invocation. The exception handler specification cannot be modified, but if the exception is to be handled by an invocation, the system pointer referenced by the exception description can be changed to locate a different program.

EXCEPTION DETECTION AND SIGNALING

Exceptions signaled by the machine are based on error conditions detected during instruction execution. The execution of a System/38 instruction is stopped and not completed when an error is detected. Various System/38 instructions that detect a particular exception do not necessarily perform the same amount of function prior to stopping execution. However, a particular instruction consistently performs the same amount of function for a particular exception. Consistent results ensure system integrity and reliability. Functions performed before exception detection are inherent to the definition of the exception or are specified in the individual instruction definition.

Once the machine detects an exception, an attempt is made in the signaled invocation to locate an exception description defined to handle that exception. If such an exception description is found, the exception handling function is performed as defined by that exception description. Otherwise, the PDEH (process default exception handler) is invoked (if one is specified).

In a similar manner, user-defined functions or subroutines can use the machine exception functions to signal the detection of an error to the current invocation or to a previous invocation. A user can employ the Signal Exception instruction to simulate the occurrence of a machine exception or a user-defined exception. This instruction causes the machine to process the exception as though it had signaled the exception.

The Signal Exception instruction can cause a new exception to be signaled or a previously signaled exception to be resignaled. The resignal option is allowed only when the signaling invocation has been invoked by the exception management functions to handle an exception.

LOCATING AN EXCEPTION DESCRIPTION

Once the exception is detected, the machine must first locate the exception description that has been defined to handle the exception. Each exception description in the signaled invocation is examined (in order of definition in the ODT) to determine whether it qualifies to monitor the signaled exception.

For machine-signalized exceptions, the invocation that is currently executing is considered to be the signaled invocation.

For user-signalized exceptions, the invocation to which the signal is to be presented must be specified. This invocation can be any existing invocation including the signaling invocation.

If no qualifying exception description is located in the signaled invocation, the process default exception handler is invoked. The process default exception handler is also invoked when the qualifying exception description attempts to immediately handle the exception with an internal subinvocation and the internal subinvocation is currently handling an exception.

Two exception handling actions can be specified that cause the machine to continue to search for another exception description. Either the exception description may be disabled, or the exception description may specify that the exception is to be resignaled to the previous invocation.

If an exception description is signaled and the signaled invocation has another exception description defined for the same exception, the machine will not attempt to locate the other exception description if the same exception occurs again. This is because the search is terminated when the signaled description is encountered even though the other description is enabled and would qualify if declared first in the ODT.

If the exception description is disabled, the machine continues to search for a qualifying exception description in the same invocation.

A compare value further qualifies the exception. When the length of the compare values specified in the exception signal and in the exception description are not identical, the length of the compare value in the exception description determines the length of the compare operation with the following considerations:

- A zero length compare value in the exception description allows the exception description to handle any exception to which the exception identification matches.
- If the length of the compare value in the exception description is less than or equal to that of the exception signal, the compare operation starts with the leftmost byte and continues to a length equal to that in the exception description. If the compare values match, the exception description is considered to handle the exception.
- If the length of the compare value in the exception description is greater than that in the signaled exception, the exception description does not qualify to handle the exception.

If the exception is to be resignaled, it is signaled to the immediately preceding invocation. If the machine locates an exception description that defines this action, the machine considers the immediately preceding invocation to be the signaled invocation and examines it for a qualifying exception handler. If no prior invocation exists in the process, the PDEH (process default exception handler) is invoked, unless the do not invoke PDEH option is specified on the Signal Exception instruction.

The Sense Exception Description instruction causes the machine to search a specified invocation for an exception description that matches an exception identifier and a compare value. When a match is detected, information about the exception description is returned (materialized).

EXCEPTION HANDLING

When the machine locates the proper exception description, the description is examined to determine how the exception is to be processed.

Ignored Exceptions

All exceptions can be ignored. If the exception description specifies that the occurrence of the exception is to be ignored:

- No record is maintained that an exception occurred.
- No invocation is notified that an exception occurred.
- No information about the exception is available to the user.

Instruction execution continues at the point where the exception occurred unless the exception was signaled by the Signal Exception instruction, and a branch option testing for the exception ignored condition is specified. Control is then passed to the specified instruction.

Deferred Exception Handling

All exceptions can be deferred. An exception description can specify that exception processing is to be deferred when a defined exception occurs. The machine records the occurrence of the exception in the exception description and saves the exception data, if specified, for later retrieval. Instruction execution continues at the point where the exception occurred unless the exception was signaled by the Signal Exception instruction, and a branch option testing for the exception deferred condition is specified. Control is then passed to the specified instruction.

The user can determine the occurrence of an exception by executing a Test Exception instruction that references the specific exception description. If the exception description has been signaled while in the deferred state, the exception-related data is presented to the user. The signaled state of the exception description is canceled, and the exception-related data is no longer available from the machine. The Test Exception instruction can specify either branching or indicator setting based on the signaled status of the exception description.

If an exception is signaled to an exception description already in the signaled state, the recording of the first exception is lost, and its exception-related data is replaced with data from the latest exception.

Immediate Exception Handling

The exception description can also specify immediate exception handling. When a defined exception occurs, the machine interrupts the execution sequence of the excepting invocation and gives control to the user-specified exception handler. The exception description can specify one of the following:

- A branch is to be executed to some instruction in the defining program.
- A subinvocation in the defining invocation is to be invoked.
- A program is to be invoked to handle the exception.

Branch Point Exception Handling

When an exception occurs, the exception description can specify that the machine is to cause control to be immediately passed to an instruction in the same invocation as the exception description.

If an exception is signaled to an invocation that specifies branch point exception handling, all following invocations except this defining invocation are destroyed. Control is then passed to the specified instruction in the defining invocation.

The defining invocation continues in normal execution as though no exception had occurred. Exception-related data is optionally available to the defining invocation through the Retrieve Exception Data instruction. Once the defining invocation is destroyed, exception-related data is no longer available from the machine.

Internal Exception Handling

When an exception occurs, the exception description can specify that the machine is to give control to an internal subinvocation that is in the same invocation as the exception description.

If the exception occurred in an invocation different from the defining invocation, all following invocations except the defining invocation are destroyed. This can occur when the following invocations resignal the exception to previous invocations until an invocation is located that immediately handles the exception. None of the implicitly destroyed invocations receive control prior to destruction. Control is then passed to the specified internal entry point.

If an exception is signaled to an invocation that specifies an internal subinvocation, all following invocations, except the defining invocation, are destroyed.

Exception-related data is optionally available through the Retrieve Exception Data instruction. Once the exception handling subinvocation is terminated, exception-related data is no longer available from the machine.

External Exception Handling

The exception description can specify that when an exception occurs, the machine is to invoke a program identified by a system pointer referenced by the exception description.

The machine invokes the program as an exception handling routine; no invocations are destroyed prior to the invocation of the exception handling routine. If no activated program (activation) exists in the process, an activation is implicitly created prior to invocation. If the invoked exception handler program has the adopt user profile attribute, the owner's user profile is used for authority verification. Normal execution in the new invocation is allowed except that the Return from Exception instruction is required to cause control to be returned from the exception handler.

Exception-related data is optionally available through the Retrieve Exception Data instruction. Once the external exception handling invocation is destroyed, the exception-related data is no longer available from the machine.

Retrieving Exception-Related Data Option

Exception-related data is optionally available. This means that when the no data retained option is specified in the exception description, the number of bytes available for retrieval field on the Retrieve Exception Data and the Test Exception instructions is set to zero. Consequently, no exception data is returned by these instructions.

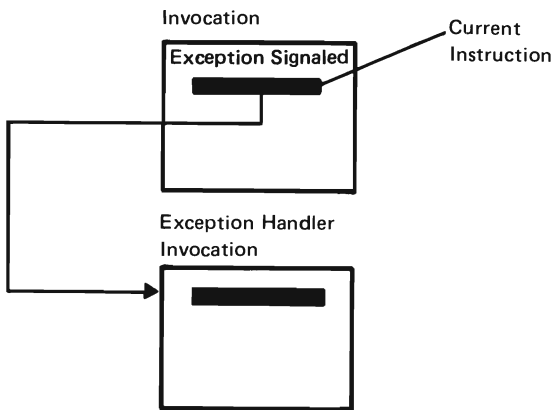
This option can be altered by the Modify Exception Description instruction and interrogated by the Materialize Exception Description instruction.

Note: An increase in system performance might be realized when exception data is not retained (for retrieval).

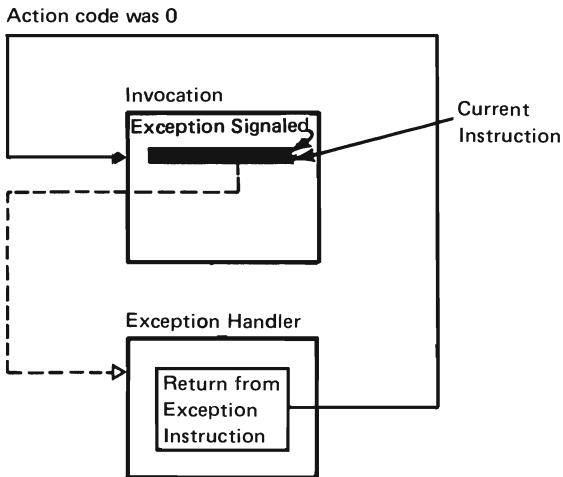
RETURNS FROM EXCEPTION HANDLING

The Return from Exception instruction terminates an internal exception handler subinvocation or an external exception handler invocation. This instruction specifies a target invocation in the PASA and an action code. Control is passed to the target invocation, and execution resumes at the instruction determined by the action code, except when a Return from Exception instruction returns to an invocation that was interrupted by an event.

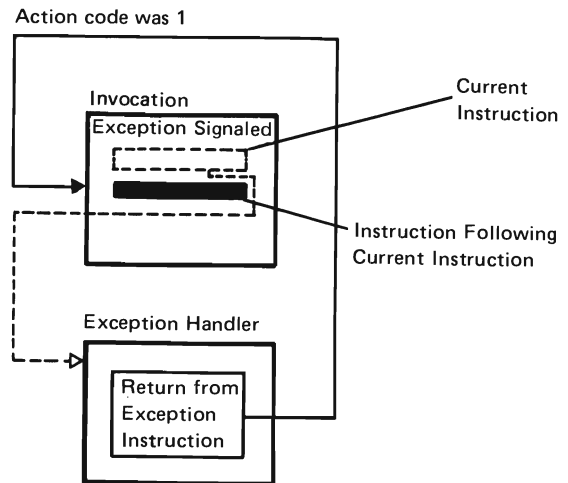
The current instruction in an invocation is defined as the instruction that caused another invocation to be created.



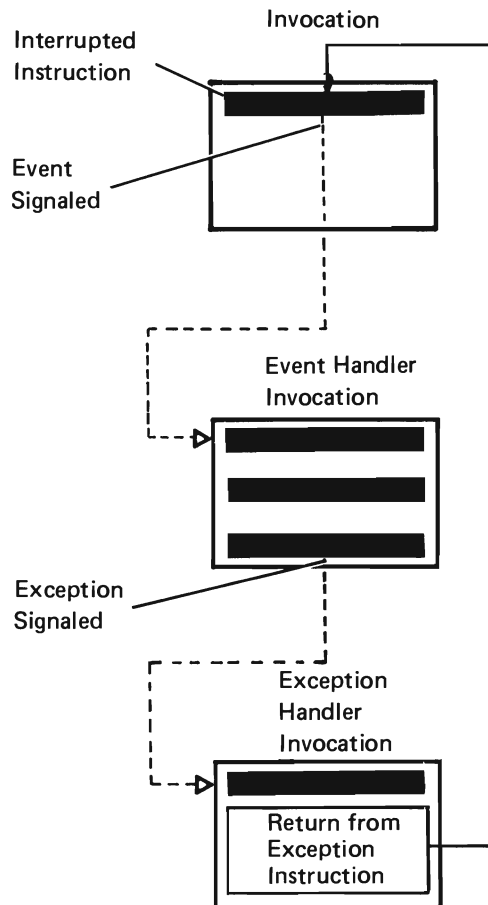
If the specified action code is 0, then the current instruction of the addressed invocation is reexecuted.



If the specified action code is 1, execution resumes with the instruction following the current instruction of the addressed invocation.



When control is returned (via the Return from Exception instruction) to an invocation that was interrupted by an event, the action code specified on the instruction is ignored, and execution continues from the point of interruption. (The interrupted instruction is completed as if no interruption occurred.)



If a significance or a size exception is signaled and the signaling is not a direct result of the Signal Exception instruction, the current instruction is completed when control is returned to the addressed invocation.

The Return from Exception instruction can be issued only from the initial invocation of an external exception handling sequence or from an invocation that has an active internal exception handler.

If the instruction is issued from an invocation that is not an external exception handler and the invocation has no internal exception handler subinvocations, an exception is signaled.

The following table shows the results of executing or attempting to execute a Return from Exception instruction:

Status of Invocation Handling Return from Exception Instruction	Return from Exception Instruction	
	Addresses Own Invocation	Addresses Higher Invocation
Not Handling exception	Invalid ¹	Invalid ¹
Handling internal exception(s)	Valid ²	Valid ³
Handling external exceptions	Invalid ¹	Valid ³
Handling external exception and internal exception(s)	Valid ²	Valid ³

¹A return instruction invalid exception is signaled. If no more internal exception handler subinvocations are active and this invocation is not an external exception handler, then the instruction cannot be issued.
²The current internal exception handler subinvocation is terminated.
³All invocations after the addressed invocations are terminated, and execution proceeds within the addressed invocation. Invocation exit programs that have been established for the terminated invocations are given control prior to proceeding within the addressed invocation.

The action code associated with the Return from Exception instruction is ignored when execution is returned to an invocation that was interrupted by the occurrence of an event. Execution is resumed at the point in the instruction where the interrupt occurred.

Whenever an invocation is terminated, the invocation count in the corresponding activation entry (if any) is decreased by one.

A Return from Exception instruction is not used or recognized in conjunction with a branch point exception handler.

EXCEPTION-RELATED DATA

Once an exception has occurred, data related to the exception is optionally available from the machine. The option is defined on the exception description and specifies whether exception-related data is to be retrieved.

The following data is available for all exceptions when the default value (binary 0) is specified:

- Exception identification
- Compare value
- Signaling invocation address
- Signaling instruction number
- Signaled invocation address
- Signaled instruction number
- Machine-dependent data identifying the machine component that caused the exception to be signaled

In addition to the preceding data, certain exception-specific data is available based on the exception being signaled. The data can include scalar and pointer data. The maximum size of exception-related data that is to accompany a signal is 32 608 bytes. The specific data for each exception is contained in the *System/38 Functional Reference Manual*.

When a user signals an exception, exception data can optionally be passed to the exception handler. This data is in addition to the standard exception data that is dependent on where the Signal Exception instruction was executed.

The exception data can be materialized by the Retrieve Exception Data instruction. This instruction selects exception-related data based on the type of exception handling (branch point, internal, or external) being performed. If, for example, internal exception handling data is selected, the data for the last exception signaled to an internal exception handler is retrieved. If no exception handler of that type was invoked, an exception is signaled.

Exception-related data cannot be retrieved once the exception handler returns control from the exception handling sequence. The exception-related data for a branch point exception handler is retained until the containing invocation is destroyed or until the exception description is signaled for another exception, whichever occurs first. The exception-related data for internal and external exception handlers is available only until the exception return is executed for the subinvocation or invocation invoked as the exception handler.

Exception-related data for deferred exceptions can be materialized by the Test Exception instruction. If the exception was signaled, exception-related data is materialized. The same information is available as for nondeferred exceptions. Exception-related data for deferred exceptions is maintained until the Test Exception instruction is executed or until the invocation that defined the exception description is destroyed.

Related data for nontested deferred exceptions is not discarded until process termination.

Exception-Related Data Option

When the no data retained option on the exception description of the ODV is specified, the number of bytes available for retrieval field on the Retrieve Exception Data and the Test Exception instructions is set to zero. Consequently, no exception data is returned by these instructions.

This option can be modified by the Modify Exception Description instruction and interrogated by the Materialize Exception Description instruction.

Note: An increase in system performance might be realized when exception data is not retrieved.

Resource Management

Resource management provides facilities to observe and control the use of and the contention for system resources. System resources include processor resources, storage resources, and system objects.

Resource management functions allow work priorities to be established and resources to be explicitly allocated. Resource management functions also enable users to communicate the characteristics of their workload to the machine, thereby permitting the machine to make efficient use of resources. These capabilities are provided through control functions and monitoring functions.

Control functions are provided through the following facilities:

- Resource management instructions specifically request the allocation of resources (locking instructions), control the overall level of work in the machine and the apportioning of resources (modify resource management controls), and provide for better utilization of storage resources (access group instructions and access state instructions).
- Process attributes affect the apportioning of resources in conjunction with resource management controls and also limit a process's consumption of resources.
- Object attributes affect the performance of programs that use the object.

Monitoring functions are provided through the following facilities:

- Resource management instructions provide data on the utilization of and contention for resources (materialization instructions).
- Process attributes provide data relative to the consumption and allocation of resources to a process.
- Machine events signal when user-specified resource utilization thresholds are exceeded.
- Exceptions indicate that a resource is not available or that use of a resource has exceeded a limit.

The following illustration shows a summary of the resource management facilities and the control and monitoring functions they perform.

Resource Management Facility	Function	
	Control	Monitoring
Resource management instructions	X	X
Process attributes	X	X
Object attributes	X	
Machine events		X
Exceptions		X



RESOURCES

This section discusses the particular resources managed by the control and monitoring functions. The subsequent section contains more detailed descriptions of the means by which these functions are provided.

Processor Resource

Processes compete for the processor on the basis of the priority process attribute. Priority does not guarantee to the various processes that they will receive relative amounts of processor resources over an interval of time; priority only serves to order the competing processes. Much of the time, processes are not competing for the processor but are waiting instead for other resources. However, priority may be used in conjunction with controls over the number of concurrently executing processes to enable an orderly sharing of the processor. The maximum processor time allowed attribute limits the total processor resource consumed by a process. To monitor consumption of the processor resource, processor time used may be materialized on a process or machine-wide basis.



Storage Resource

Machine storage is composed of various physical storage media, differing from each other in capacity and access speed. These media fall into two levels of storage, main storage and auxiliary storage. Instruction execution requires that instructions and referenced data be present in main storage. Processes compete for storage resources and for the access mechanisms by which data is transferred between media. The machine automatically manages this allocation and data transfer on a demand basis. However, this management entails machine overhead, involves delay to the requesting process, and directly affects the relative performance of competing processes. Therefore, resource management provides a variety of tools for communicating to the machine additional information affecting the management of the storage resource.

Multiprogramming Level Control

The machine provides facilities for limiting the number of processes that concurrently compete for the machine resources. These facilities can directly limit the contention both for main storage and for access mechanisms. This enables the user to prevent situations in which this contention could adversely affect the total work accomplished by the system. The machine maintains the number of data transfers to and from main storage to assist the user in determining the appropriate limits.

Main Storage Control

Main storage pools are the primary means for apportioning the main storage resource among processes. The machine provides a means for dividing the available resource into pools of designated sizes and for assigning processes to these pools. The machine then satisfies a process's requirements for main storage from the assigned pools. Information regarding the amount of data transferred to and from each pool is provided.

Data Transfer

Several functions are provided to enable more efficient transfer of data between levels of storage, and for providing the machine with explicit information on when objects are needed for instruction execution and when they are no longer required. These functions enable better use of resources both by reducing contention for physical access mechanisms and by making main storage resources more readily available for other uses.

The access state of an object refers to the speed with which it can be accessed and is, thus, determined by the characteristics of the storage media on which it resides. The Set Access State instruction provides a mechanism for altering the access state of an object in anticipation of subsequent use of the object. This instruction can be used to inform the machine that one or more objects will be needed in main storage for instruction execution, or that one or more objects are no longer needed and should be placed on a slower medium. The instruction enables the machine to efficiently transfer the object as one or more large blocks and/or to make the associated main storage available for other uses.

Access groups provide a mechanism for physically collecting objects to more efficiently use the storage facilities. They enable the machine to transfer a set of objects as a single, larger object. Additionally, by designating a process access group, the user can identify to the machine a set of objects, which need not be present in main storage when the associated process is not active and which the machine can efficiently transfer to and from main storage.

Further control can be applied at the object level. Through the block transfer indicator contained in the performance class attribute of an object, the user can specify that a reference to an object will automatically cause a block of data (whose size is machine- and object-dependent) to be made available in main storage. Space objects, data spaces, and data space indexes, which typically represent a large proportion of the data transferred, can be assigned to a particular unit of auxiliary storage when created through the unit parameter in the create template.

Auxiliary Storage Control

If the machine typically runs sufficiently below a condition of fully allocated auxiliary storage and if dynamic requests for space can always be honored, contention for space on auxiliary storage normally does not occur. The machine signals an event if a process exceeds the limit on the amount of temporary storage that can be allocated by the process. (The limit is set during process initiation.) Because permanent objects normally exist beyond the life of a process, control of the allocation of permanent space is enforced through the user profile that owns the object, rather than through the process itself. Finally, the machine signals the auxiliary storage threshold exceeded event whenever the amount of available (unallocated) space falls below a user-specified threshold.

The machine makes available to the user the amount of unallocated space existing on each unit of auxiliary storage as well as how many data transfers occur to and from each unit.

System Objects

A system object can be shared by processes if these processes have no requirements to complete a specific series of instructions on that object. That is, the machine guarantees the correct execution sequence of individual instructions, but not of a series of instructions. Because the execution of a specific series of instructions is frequently required, the machine provides instructions for locking objects to processes. A lock on an object determines which locks on the object may concurrently be held by other processes and which types of access (materialize, modify, or control) may be granted to other processes executing instructions against the object. Thus, the locking instructions provide:

- A mechanism whereby processes can observe a protocol for synchronizing access to objects
- A means to prevent other processes, not observing a lock protocol, from accidentally or intentionally accessing an object temporarily not available to them

CONTROL AND MONITORING FUNCTIONS

This section describes the various functions provided by resource management.

Multiprogramming Level Control

Multiprogramming level (MPL) controls can limit the number of concurrently executing processes. When used in conjunction with the time slice process attribute, priority process attribute, and the process access group, multiprogramming level controls provide the primary means for sharing of resources.

Basic Concepts

In the absence of resource management controls (MPL rules), a process may be in one of the following states: instruction wait state, suspended state, or active state.

A process is placed in the instruction wait state in the following circumstances:

- When the Wait on Event instruction is issued
- An unsatisfied lock exists (synchronous wait option)
- When the Set Cursor instruction is issued for update or modify and the affected data space entries are locked by another process
- When a Dequeue instruction with the wait option is issued

A process is in the suspended state if a Suspend Process instruction has been issued against the process. For purposes of this discussion, the suspended state and the instruction wait state are treated as equivalent. When a process is not in one of these states, it is in the active (executable) state. MPL controls allow the user to specify the number of processes that can concurrently be in the active state. When this number is exceeded, one or more processes are placed in a fourth state called the ineligible state (according to MPL rules).

To provide a further level of control, processes can be assigned to MPL classes. For each MPL class, the user can specify a maximum MPL; that is, a maximum number of processes, assigned to a class, which can be concurrently executing. Thus, a process can be constrained both by its class MPL limit and by the machine-wide MPL limit. Through the use of these controls, a user can specify, for example, that no more than six processes are to execute concurrently in the machine and that no more than four processes in class A and three in class B are to execute concurrently.

Machine-wide and class MPL controls are established by the Modify Resource Management Controls instruction. Processes are assigned to a particular class through the MPL class process attribute. Before describing the actual rules involved in MPL enforcement, two additional attributes (priority and time slice) must be described.

The priority attribute is a process attribute that was briefly described in connection with the processor resource. Processes can be ordered by priority when they are:

- Waiting for the processor or for certain serialized internal machine functions
- Waiting for a lock, a locked data base record, or a message on a queue
- In the ineligible state

The time slice attribute is a process attribute that specifies an amount of processor time during which a process can execute instructions before being subjected to the application of MPL rules.

MPL Rules

When a process is placed in the wait state or when it terminates, the machine-wide and class MPLs (multiprogramming levels) are decremented. (The Dequeue instruction can be made exempt from this rule when a minimal wait is anticipated.) Processes in the ineligible state are examined in priority sequence to see whether any can be made active without exceeding the machine-wide or class MPL maximum values. If so, the selected process is made active.

When a process is initiated or leaves the wait state, resource management determines whether the process can be made active without exceeding the machine-wide or class MPL limits. If so, it is made active; if not, it is placed in the ineligible state. It does not preempt processes of a lower priority currently in the active state.

When the time slice amount for a process expires, signifying that it has received a specified amount of processing unit time, resource management determines whether there is a process of equal or higher priority in the ineligible state that could be made active according to the machine-wide or class MPL limits; if so, the current process (the process whose time slice expired) is placed in the ineligible state, and the selected process is made active; if not, the current process is allowed another time slice. Thus, processes of equal priority share the processor according to the MPL rules. A higher priority process preempts a lower priority process only when the time slice for the lower priority process has expired.


Processing When Entering or Leaving the Active State

When a process is in the active state, instructions and data are needed in main storage for execution. When the process leaves the active state, portions of this process may still be required by other currently executing processes (for example, programs shared by processes). Other unshared data will not be needed until the process reenters the active state. Normal management of storage by the machine eventually makes the main storage occupied by this unshared data available to other processes. However, the machine enables the user to expedite this process through use of a process access group.

A process access group is an access group that is designated as a process attribute on the Initiate Process instruction. It contains a set of objects used almost exclusively by the process with which it is associated. This set of objects is transferred from main storage to auxiliary storage (purged) when a process leaves the active state. This same set of objects is then transferred back to main storage when a process reenters the active state.

A process can leave the active state to enter either the wait state or the ineligible state. During these two state transitions, two process attributes, instruction wait access state control and time slice end access state control, designate whether the process access group is to be transferred upon leaving the active state. Additionally, instructions that place a process in wait state specify whether the access group is to be purged. (These same instructions also specify whether to transfer the process access group to main storage when the wait is completed.) The purge of the access group occurs during the wait state only if both the process and the instruction indicate that the purge is to occur.

The purge indicators are thus used to indicate that the wait could be long enough so that the overhead of freeing main storage resources is justified. Whenever the process access group is purged, the time slice amount for the process is reset so that when the process next enters the active state, the process will be allowed the full time slice amount. If the purge does not occur, the time slice is not reset.




Ineligible Threshold

If a large number of processes are in the ineligible state because of machine-wide or MPL class controls, then the machine may be overcommitted or one or more MPL limits may be set to inappropriate values. The user can specify threshold values, machine-wide and by MPL class, which, if exceeded by the number of processes in the ineligible state, cause the ineligible state threshold exceeded event to be signaled. The threshold values are specified by the Modify Resource Management Controls instruction.

Monitoring MPL Activity

The Materialize Resource Management instruction provides data on both a machine-wide and MPL class basis, on the total number of processes, on the number of processes in the ineligible state, and on the number of state transitions.

Storage Resource Functions



This section describes the functions provided to make better use of storage resources.

Main Storage Pools

Main storage pools provide a way for competing processes to share main storage resources. The Modify Resource Management Controls instruction can be used to partition main storage into several pools. The machine then reserves the designated amount of main storage in the specified pools. Processes are assigned to specific pools by the main storage pool process attribute. When instruction execution for a process requires that data be placed in main storage, resources are allocated from the assigned pool.

Initially, the main storage resource is assigned to the machine main storage pool. Main storage required for internal machine functions is allocated from this pool. The user can then assign portions of the remaining resource to other pools; however, a machine-dependent minimum amount of main storage must always remain in this pool. Certain objects (for example, operating system programs) can be treated as machine objects shared by processes. The user may force such objects to be allocated main storage from the machine main storage pool through the main storage pool selection indicator in the performance class attribute. The Set Access State instruction, when used to request transfer of data to main storage, can specify that storage is to be allocated from a particular pool. In the absence of such specification, main storage is allocated from the pool assigned to the requesting process.

Some objects, such as those objects required to handle an exception, might be required to be in main storage for only a short time. When these objects are paged into the machine pool or any storage pool shared by users, they might displace certain data needed for normal processing. To minimize this condition, the user can specify a transient pool and designate (via the performance class attribute) objects as transient. Then, when these transient objects are paged into main storage, they are allocated storage from the transient pool. This way, transient objects can replace other transient objects rather than those objects used for normal processing.

Main storage pools, however, do not enable the user to precisely manage the contents of main storage. Rather, the intent is to provide the user with sufficient control to divide the resource among processes and categories of work so that the desired work priorities, throughput, and response time objectives can be achieved.

Access Groups

Access groups provide a means of collecting temporary system objects to enable more efficient use of storage resources. Members of an access group are located next to other members on the auxiliary storage media so that they can be efficiently transferred as a single, large object. Access group transfers may be initiated explicitly through the Set Access State instruction, or implicitly when a process is going to or from the active state if the access group was identified as the process access group. Reference to objects within an access group does not require transferring the entire access group; only the required data need be transferred.

An access group is created by the Create Access Group instruction. An object can be placed in an access group by specifying the access group operand in the create template of the object when it is created. An object is removed from an access group when the object is destroyed. The space that was occupied by the object can be reset by the Reset Access Group instruction. This instruction effectively compresses the space occupied by the access group.

The access group itself is destroyed by the Destroy Access Group instruction. But, an access group cannot be destroyed until all objects are removed. Finally, the Materialize Access Group instruction can be used to materialize the attributes of an access group and a list of the objects contained in the access group.

Set Access State

The Set Access State instruction is used to temporarily alter the access state of an object (that is, the access speed implied by the storage level) in anticipation of the subsequent use of that object. The options available and uses for the instruction are dependent on the storage configuration of the machine. In all machines, however, an object (or portion of an object) can be designated as required in main storage. The instruction implies no guarantee of how long the access state specification is in effect, as this is partially a function of resource demand by other processes in the machine. When requesting that data be available in main storage, the user may optionally designate a main storage pool from which storage is to be allocated for the object.

The Set Access State instruction optionally can cause a pseudo retrieve object operation to be performed from auxiliary storage. This means that the instruction causes a specified object to be placed in main storage without regard to the contents of the object. When this option is used to retrieve an object, access to auxiliary storage is reduced.


The Set Access State instruction can also cause an object to give to another object the space it occupies in main storage. For additional information about the options for this instruction, refer to the *System/38 Functional Reference Manual*.



System Object Attributes

Certain system object attributes that are specified when the object is created (but not, in general, modifiable), affect the performance of the processes that use the object. The performance class is the principal means of specifying these attributes. Fields within the performance class, in turn, specify the following:

- The main storage pool selection attribute causes the object to be allocated main storage from the machine pool.
- The transient attribute causes the object to be allocated main storage from the transient storage pool. If a transient storage pool is not specified, the object is allocated main storage depending on the status of the main storage pool selection attribute.
- The block transfer attribute causes the machine to make available in main storage a larger portion of the object than is required for execution of the immediate instruction. This is useful when subsequent instructions will reference other portions of the object. The size of the block is machine- and object-dependent.



The unit parameter can be specified only for spaces, data spaces, and data space indexes. The user then has some control over allocating these objects to specific auxiliary storage units. This capability makes possible the balancing of the data transfer load among access mechanisms. (Normally, objects are allocated to a unit according to internal algorithms.)

Storage Limits

The machine enforces certain limits on the use of auxiliary storage. The user is thus prevented from accidentally or intentionally overcommitting storage so that normal operations cannot proceed.

The process auxiliary storage limit process attribute provides a means to limit the total space that can be allocated to objects created by the process with the temporary attribute. When this limit is exceeded, an event is signaled.

The permanent auxiliary storage limit is a user profile resource authorization that limits the total space that may be allocated to permanent objects owned by the user profile. When this limit is exceeded, an exception is signaled to the requesting process.

When the amount of unallocated auxiliary storage falls below the value specified by the auxiliary storage threshold, the auxiliary storage threshold exceeded event is signaled. This value is set by the Modify Resource Management Controls instruction. No exception is signaled.

Machine storage limit exceeded is an exception that is signaled when an instruction cannot be completed because of insufficient auxiliary storage space.

Process Attributes

Process attributes are specified in the process definition template. These attributes are used by the Initiate Process instruction and modified by the Modify Process Attributes instruction. A subset of the process attributes, which follows, applies to resource management functions. Most of these attributes have already been described with the multiprogramming level functions (refer to *Multiprogramming Level Control* earlier in this chapter).

- Time slice.
- Priority.
- Instruction wait access state control.
- Time slice end access state control.
- Process access group.
- Process multiprogramming level class ID.
- Storage pool ID (described in connection with main storage pools).
- Total processor time allowed specifies the amount of processor time that a process may consume. When this limit is reached, an event is signaled.
- Maximum temporary auxiliary storage allowed limits the total space that can be allocated to objects with the temporary attribute.
- Default time-out interval applies to instructions that can place a process into a wait. It specifies (in the absence of an explicit specification in the instruction itself) the maximum time the process is to remain in the wait state. If this limit is exceeded, an exception is signaled. (Note that the use of the time-out function is a means to break deadlocks arising from conflicting lock requests by two or more processes.)

Process external existence state describes the current (at the instant of the materialize) state of the process: active, ineligible, suspended, or in a wait.

The following values can be materialized by the Materialize Process Attributes instruction, but they cannot be modified.

- Process auxiliary storage used.
- Total processor time used.
- Number of locks currently held.
- Number of transitions into ineligible wait state.
- Number of transitions into instruction wait state.
- Number of transitions into ineligible wait state from instruction wait state.
- Number of synchronous read operations to the data base portion of auxiliary storage.
- Number of read operations from auxiliary storage that are not associated with the data base.
- Number of write operations to auxiliary storage.
- Additionally, the modifiable attributes previously described may be materialized.

SYSTEM OBJECT LOCKS

System object locks are used to:

- Ensure the integrity of system objects and operations involving them when they are shared by two or more processes
- Provide for the allocation of system objects to processes
- Provide for the transfer of the allocation of system objects from one process to another process

The Lock Object, Unlock Object, and Transfer Object Lock instructions allow direct control of the use or allocation of system objects. Instructions that operate on a system object ensure that the use of that object is not in conflict with any lock held by another process; if this conflict exists, the operation is not performed and an exception is signaled. A process can also lock an object in a way that guarantees the use of the object without conflict. The Lock Object instruction provides the capability to wait for other processes to remove any incompatible locks on an object.

Types of System Objects That Can Be Locked

Locks control three classes of operations on objects: materialize, modify, and control.

The lock states guarantee a process the ability to perform various combinations of these classes of operations while prohibiting certain combinations to other processes. For more information on locks and classes of operations for each instruction, refer to *Lock Enforcement* in the *System/38 Functional Reference Manual*.

All temporary and permanent system objects can be locked by a process with one or more of the following lock types.

Lock Shared Read (LSRD): This lock guarantees the lock holder the ability to perform materialize operations on objects. Other users may materialize or modify but not control the object.

Lock Shared Read Only (LSRO): This lock guarantees the lock holder the ability to perform materialize operations on the object. Other users may only perform materialize operations.

Lock Shared Update (LSUP): This lock guarantees the lock holder the ability to perform materialize and modify operations on the object. Other users may materialize and modify but not control the object.

Lock Exclusive Allow Read (LEAR): This lock guarantees the lock holder the ability to perform materialize and modify operations on the object. Other users may only materialize the object.

Lock Exclusive No Read (LENR): This lock guarantees the lock holder the ability to perform any operation on the object. Other users may not operate on or lock the object.

Lock Request Granting Algorithm

The basic allocation rules are:

- Only one process can hold an LENR lock on a system object at any time. No other process can hold any other lock on that same object.
- Only one process can hold an LEAR lock on an object, but other processes can simultaneously hold LSRD locks on that object. LEAR is not compatible with the LSUP, LEAR, LSRO, or LENR locks held by other processes.
- One or more processes can hold an LSUP lock on an object while other processes are also holding LSRD or LSUP locks on that object. LSUP is not compatible with the LSRO, LEAR, or LENR locks allocated to other processes.
- One or more processes can hold an LSRO lock on an object while other processes are holding LSRO or LSRD locks. An LSRO lock is not compatible with the LSUP, LEAR or LENR locks allocated to other processes.

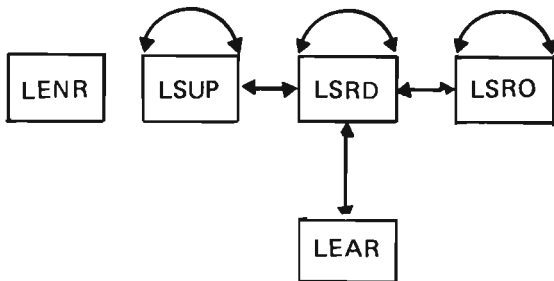
- One or more processes can hold an LSRD lock on an object while other processes are holding LSRD, LSRO, LSUP or LEAR locks. An LSRD lock is not compatible with an LENR lock held by any other processes.
- A process can hold object locks of one or more lock types on an object if the previous rules are not violated. Therefore, a process can be allocated all five types of locks on an object if, and only if, it is the only process that holds any object lock on the object.

Table of Lock States: The lock granting rules are summarized in the following table:

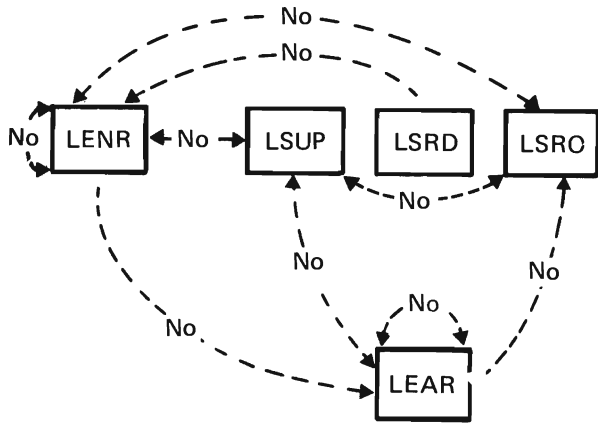
Lock State Requested	Lock States Unavailable to Other Processes	Guaranteed Operations to This Process	Operations Allowed to Other Processes
LENR	LENR, LEAR, LSUP, LSRO, LSRD	Materialize, modify, control	None
LEAR	LENR, LEAR, LSUP, LSRO	Materialize, modify	Materialize
LSUP	LENR, LEAR, LSRO	Materialize, modify	Materialize, modify
LSRO	LENR, LEAR, LSUP	Materialize	Materialize
LSRD	LENR	Materialize	Materialize, modify

A maximum of 57 344 locks, implicit locks, and data base entry locks can exist at any one time.

Lock Coexistence Graph: An arrow between two lock types indicates possible coexistence by different processes.



Lock Exclusion Graph: An arrow between two lock types indicates a conflict by different processes.



A lock may be granted or refused, but a lock can never preempt an existing lock on a system object.

Any combination of lock types is allowed when only one process is holding locks on a system object. For example, the process can lock a system object with all five lock types. The process can unlock each lock in any sequence. A process may also lock an object multiple times with the same lock type. Each of these multiple locks must be individually unlocked.

Two or more processes can each hold more than one lock type on a system object as long as these lock types are allowed to coexist. The processes can unlock the more restrictive lock at a later time when only the less restrictive lock type is needed.

The requesting process is put into the lock-wait state if the wait option is specified by the requesting process when a lock being requested conflicts with a lock held by another process. Otherwise, an invalid lock state exception is signaled.

When multiple processes are waiting for lock requests to be granted, the lock request granting algorithm is as follows:

- For all processes that are requesting a lock on a system object, dispatch the processes in their process priority if there is no lock holder on the system object.
- If there are one or more lock holders on the system object, dispatch the processes to determine whose requesting lock types can coexist with the existing locks on the system object. Dispatching is based on priority.

Figure 4-2 shows an example of a process unlocking an object for which other processes are contending. When process P1 deallocates system object A by unlocking its LEAR lock, processes P3, P4, and P5 are dispatched. However, process P5 resumes waiting because the LENR lock is not allowed to coexist with any other lock type. The machine attempts to grant the lock request to the process (P3 or P4) with the higher dispatching priority. The process not granted its lock request resumes waiting.

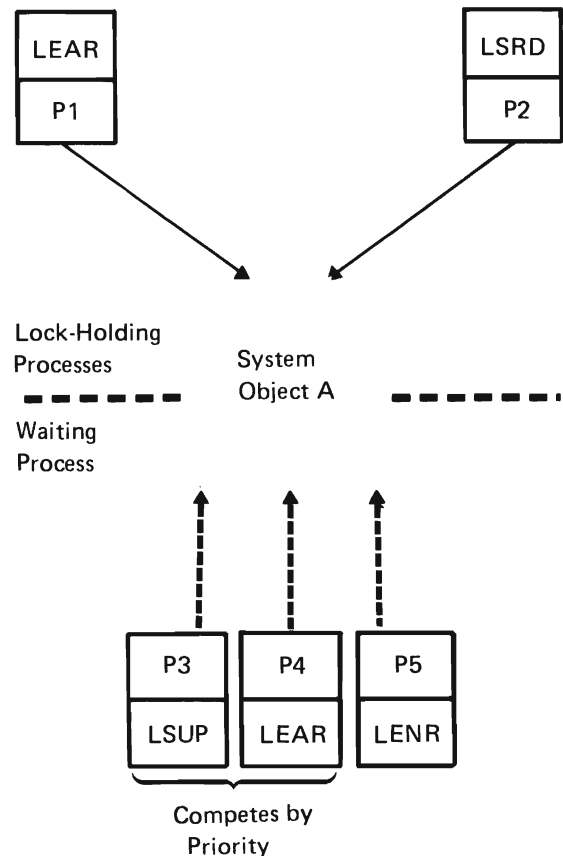


Figure 4-2. Multiple Lock Holding and Waiting Processes

Sharing Data Within a System Object

There is no lock enforcement on bytes or strings of bytes within a space object or a space associated with a system object. Even though a space object is locked with an LENR lock, any process can still operate on the data in the space. In this case, the user must follow his own conventions in order to serialize operations.

When the entries in a data space system object are to be updated, the cursor effectively locks them.

Implicit Locks

Certain instructions cause locks to be implicitly applied on system objects. These implicit locks represent an implied allocation of these objects to a process. For example, the Activate Cursor instruction causes an implicit lock to be applied to the cursor. This lock is implicitly removed when the cursor is deactivated. The user cannot explicitly unlock an implicit lock with the Unlock Object instruction. Implicit locks, as well as user-applied explicit locks, can be materialized to determine how the system objects are allocated among processes.

Transferring Locks

Use the Transfer Lock instruction to transfer a lock on a system object from its present lock holder to another process.

Transferred locks must not create a conflict with other process locks. For example, if a process has both an LSRD and an LENR lock on the same object, neither of these two locks can be transferred because that would create a violation of the locking rules. Because the transfer of locks is performed one lock at a time, a request to have both locks transferred by one instruction is not allowed. In this case, unlock the less restrictive lock (LSRD) and transfer only the most restrictive lock. Transferring locks is logically equivalent to the sending process unlocking the object, one lock at a time, and the receiving process locking it without having to wait.

Locks can also be transferred by the Initiate Process instruction from the initiating process to a new process. Implicit locks cannot be transferred.

Locking a Space Location

The Lock Space Location instruction applies a symbolic lock to a specific space location. However, even though the location is locked, this does not prevent any byte operation from referencing or modifying that location, nor does it prevent the space from being extended, truncated, or destroyed. Otherwise, space location locks follow the normal locking rules with respect to conflicts and waits.

Unlocking a Space Location

The Unlock Space Location instruction removes locks from a space location. The locks must be held by the process that issues the instruction. Otherwise, an exception is signaled. When multiple locks of the same lock state for the same space location need to be removed, this instruction must be issued a number of times up to the number of locks held for the space location.

The space location need not exist when this instruction is issued, but the space pointer must be a valid pointer.

Materializing Locks

The Materialize Selected Locks instruction causes the locks (for either objects or space locations held by the process issuing the instruction) to be materialized.

The Materialize Object Locks instruction causes the current lock status of either an object (identified by a system pointer) or a space location (identified by a space pointer) to be materialized.

The Materialize Process Locks instruction causes the lock status either of a process identified by a system pointer or of the process that issues the instruction to be materialized.

The Materialize Data Space Record Locks instruction causes the current lock status of a data space record to be materialized.

The Materialize Process Record Locks instruction causes the current allocated data space record locks held by a process to be materialized.

Unlocking System Objects

The process that is the current holder of a system object lock is allowed to unlock that system object lock. Any number of the locks held by a process on a system object can be unlocked with one unlock instruction. Each removed lock reduces by one the lock count of that lock type held by that process. Therefore, if the process had locked the system object with one lock type more than once, the same number of unlock requests must be issued to remove all locks of that type. When a process is holding a lock and that process is terminated, then all objects locked by that process are released.

Implicit locks held by any process cannot be unlocked with the Unlock Object instruction. These locks are implicitly unlocked according to conditions under which the instruction applies these locks. Implicit locks and user-applied locks that are held by a process are always unlocked during the termination of that process.

When an object is destroyed, all locks for that object are released.

Deadlock

Deadlock occurs when two or more processes are waiting to use an object without which they cannot continue, but that object is presently locked to one of the other waiting processes. The processes are deadlocked because each is waiting for one of the other processes.

Approach to Reducing Deadlock Situations

In deadlock detection, an algorithm is used to determine whether a deadlock between two or more processes would occur when a process goes into a wait for a lock.

In deadlock prevention, if waiting for a certain lock could result in a deadlock, the lock is not granted, the process does not wait, and therefore the deadlock is prevented. This implies deadlock detection at allocation time, but does not imply that the machine is deadlock-free.

Deadlock-free means that deadlocks are not only detected and prevented, but all processes are allowed to proceed to completion without getting into deadlocks and without knowledge of the existence of other processes during their execution.

The basic approach for deadlock prevention is to reduce the possibility of deadlocks by requiring System/38 programs to follow certain programming conventions and by being able to detect deadlocks.

Because sharing of objects is allowed, only processes that require exclusive use of objects can get into deadlocks. By observing certain programming conventions, the possibility of deadlock is reduced in what is basically a demand allocation machine.

The following two programming conventions do not imply preallocation of all resources required by a process; they apply only to the known resource requirements at the time the resources are needed. Each convention has the same effect in reducing the occurrence of deadlocks due to sequence of resource allocation; the second convention requires an allocation order be established and observed by all programs for all system objects.

- All known system objects that are required before a process can continue its processing are allocated collectively; if any one allocation request is not granted, all system objects that are already allocated are deallocated. The request for allocation is repeated at a later time until all system objects are collectively allocated.
- Before a process can continue processing, all known and required system objects should be allocated in the same order by all programs. This allows a process to wait for the unavailable system object.

The use of these programming conventions can reduce deadlock situations and detect deadlocks due to sequence of resource allocation. This approach, however, does not ensure that the machine is deadlock-free.

Deadlock Due to Sequence of Lock Application

The simplest case of interlocking deadlock is shown in Figure 4-3. Process P1 is holding object B and waiting for object A; at the same time, process P5 is holding object A and waiting for object B. Because preemption does not occur, processes P1 and P5 are deadlocked with objects A and B.

Notice the deadlocked loop that exists from P5, to Object A, to P1, to Object B, and back to P5. This loop represents the deadlock between processes P1 and P5 over objects A and B.

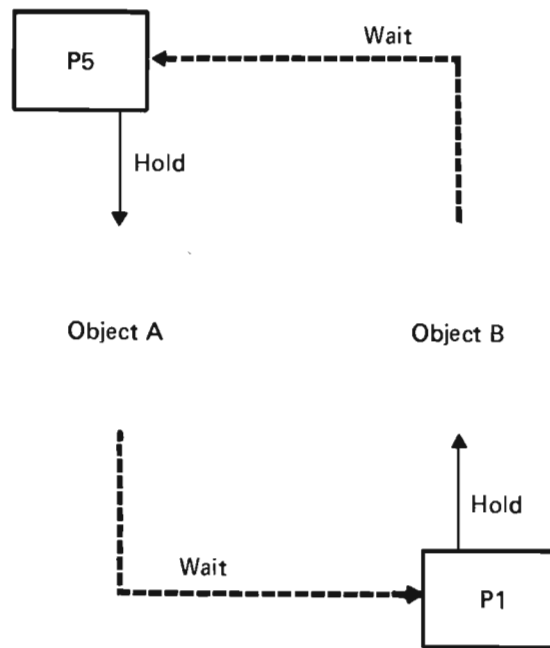


Figure 4-3. Deadlocked Processes (P1 and P5)

Figure 4-4 shows that for a process to be deadlocked with another process or processes, the process must be simultaneously holding a lock on a system object as well as be waiting with a lock request for another system object. Therefore, in Figure 4-4 processes P2, P3, P4, P7, and P8 cannot be deadlocked with other processes because, individually, none of them is both holding and waiting for system objects. However, two deadlocked loops do exist:

- P1, object B, P5, object A
- P1, object D, P6, object B, P5, object A

The first loop **1** is the simple case; the second loop **2** involves the deadlock of three processes with three system objects.

Deadlock Detection and Resolution

Because resources are allocated either when needed or on demand, a deadlock situation is not known until it occurs.

Deadlock situations can be resolved when a lock-wait time-out occurs. At this time, the wait by a process for a lock is released.

When a deadlock occurs, isolate the source of the deadlock. Check all processes for the necessary conditions for a deadlock (a process that is simultaneously holding a lock and waiting for another one). Processes that satisfy the necessary conditions should be further checked with the process-object sequence test as illustrated in Figure 4-4. If one or more closed loops are found, a deadlock has occurred. Terminating any one of the processes in each loop eliminates the deadlock situation.

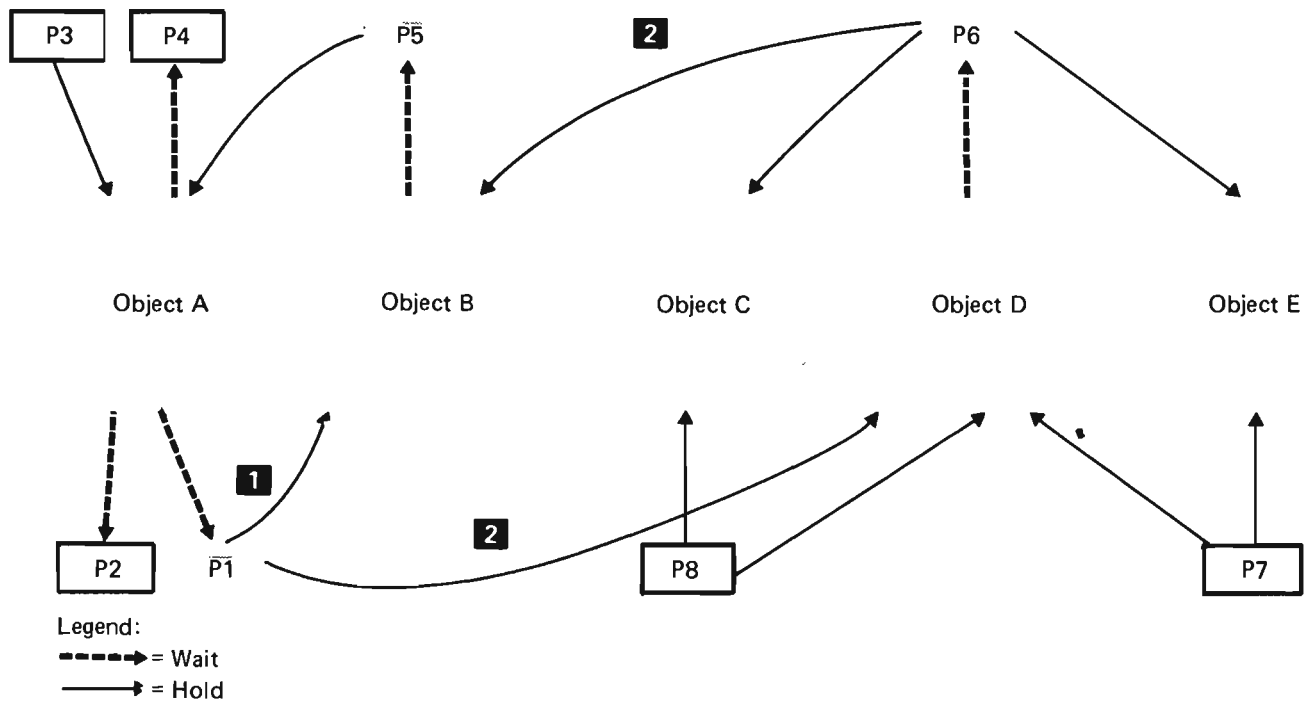


Figure 4-4. Deadlocked Processes (P1, P5 and P6)



Data Base Management

The data base management functions enable the user to store, manage, and use data.

Data is made up of individual data fields. Groups of data fields are stored as entries (records) in objects called data spaces. An entry is the basic unit of a data space with each entry having the same format as all other entries in that data space. Entries can be inserted, retrieved, updated, and deleted. Through use of a cursor, the user can specify a view (definition) of the entry to be a subset of the actual fields, to be in a different order, or to have different attributes than the actual entry in the data space.

When specifying views for retrieving data, the following additional functions are supported. The user can specify:

- Selected entries (selection criteria) to view.
- A view in the form of a derived entity (for example, multiply the contents of a field by 10).
- A joined view that allows mapping and/or deriving of entities from multiple data space entries into a single logical view.
- Group-by functions that allow clustering of one or more data space entries into a single image the user can view. The image is comprised of cumulative results of specific fields from each entry (for example, return the sum of salaries for a group of employees, or return the maximum salary of a group of employees).

Through use of the data space index, the user can cause the entries to appear to be ordered differently than they actually are. Data spaces and data space indexes can be shared among concurrent processes.

MAJOR DATA BASE OBJECTIVES AND CHARACTERISTICS

The major data base objectives are to:

- Provide the basic functions for the management of data (the ability to insert, retrieve, update, and delete data).
- Provide late bound views of data to meet the changing needs of application programs. Late bound view attributes include the organization, location, and attributes of the data.
- Provide multiple views of data. Different applications can view the same data differently without duplication of the data. Multiple views include multiple orderings of the same data as well as transformations between the stored representation and the user's view of the data.
- Provide representation independence of managed data through the use of symbolic addressing of data and transformation capabilities. This enables the user to be independent of the internal representation of data.
- Provide security of managed data by accessing the data only via interfaces that are part of the architecture and through support of system authorization.
- Provide integrity of managed data by storing and enforcing the user-defined description of individual fields in the managed data. Data integrity is also provided via interlocks for shared access to managed data in a multiprogramming environment and through automatic recovery procedures in case of machine failure.

DATA BASE OBJECTS

The following system objects are used by the data base:

- **Data space**—An object containing entries that can be inserted, updated, retrieved, or deleted by a program. An entry is an ordered collection of fields, each with a type and length. All entries in a data space are homogeneous; that is, all entries in a data space have identical field definitions.

Entries in a data space are positioned by arrival sequence with each entry having a permanently assigned ordinal number identifier. Deleted entries continue to occupy space because the system will not reuse a deleted entry until a user explicitly uses that space.

- **Data space index**—An object that can logically order data space entries independently of their physical order in the data space. A data space index can be created over more than one data space, and more than one data space index can exist for a data space. The creation of a data space index is similar to, but much more powerful than, a sort of the data. With a data space index, minimal duplication of data occurs, and the new ordering is permanently maintained by the system. Changing a data space entry results in automatically updating all data space indexes that reference that data space.

- **Cursor**—An object that provides access to data space entries. A cursor is not only the user's interface to the data base but, in addition, it contains the definition of the user's view of the data. This definition directs the mapping that takes place to and from the user's buffers. The user's view of the data can be made up of a subset of the fields, and with different field attributes (data type and length) from those in the data space entry. The mapping, conversion, selection, joining, grouping, and derive operations are done implicitly from the information contained in the cursor object.

Cursors can provide access to data space entries in the same order as they are stored in the data spaces or through the logical ordering provided by a data space index. Cursors can be created over multiple data spaces. The use of cursors permits concurrent access to the same data space by more than one process.

A cursor has two states, activated and de-activated. The creation of a de-activated cursor effectively prebinds a set of data (data spaces), a logical ordering of the entries (a data space index), and the user's view of the various entries (data mapping). The activation of a cursor binds these elements to a specific process and maintains information specific to that process, (for example, the location of the currently addressed entry). An activated cursor can be used only by the process that caused its activation.

USING DATA BASE FUNCTIONS

Creating a Data Space

A data space is created by building a data space template and then issuing a Create Data Space instruction. The data space template describes the entry format for the data space, the number of fields, their order within a data space entry, and the attributes (type and length) of each field. Optionally, a default values entry can also be provided. These values are used for fields not present in the user's view of the entry when an entry is being inserted into the data space.

Creating a Cursor

A cursor is created by building a cursor template and then issuing the Create Cursor instruction. The cursor template specifies the data to be accessed through the cursor (the data spaces), whether the user will see the data as actually ordered or as seen through a data space index, and specifies the user's view of the various data space entries.

Associated with each data space is a mapping template that defines the user's view. The user can specify both an in- and an out-mapping for each data space. The in-mapping is used for inserting and updating entries, while the out-mapping is used for retrieve operations. The in- and out-mapping can be either identical or different. The in- or out-mapping specification can be identical to the physical entry; or it can specify a subset of the fields, a different order of the fields, or different attributes for any or all fields. Through proper definition of the mapping attributes, the resultant cursor can be created so that individual fields can either be retrieved and not inserted, or be inserted and not retrieved.

In conjunction with out-mapping, derived field processing can be performed on fields from the entry (for example, multiply a field by 5).

Optionally associated with each data space is a selection template that filters out entries that should not be addressed via the specific cursor. Without this selection template, each entry in the data space is available for retrieval. With selection criteria specified, only those data space entries that satisfy the selection criteria are retrievable and available for viewing. This selection criteria is associated only with data retrieval.

A user can specify to join data space entries when creating the cursor. Joining is the process of using multiple data space entries as sources for fields in a single resultant 'joined entry.' The data spaces associated with the cursor are the sources of the data space entries to be joined. Each data space contributes an entry, with fields extracted as defined at cursor creation. The resulting image, or 'joined entry,' is then returned and viewed as a single entry. The joining criteria is specified in a cursor creation time template. Equal joins are allowed.

Join is supported only for retrieval operations (read only join).

The user can specify a join that joins multiple data spaces to one data space (many to one) and a join that joins multiple data spaces to multiple data spaces (many to many).

The user can optionally specify group-by operations when creating the cursor. Group-by allows clustering of one or more data space entries into a single image for viewing.

Group-by can be specified for both join and non-join cursors. Group-by can also be specified for cursors using data space indexes and for those without.

Group-by is performed only through the Retrieve Sequential Data Space Entries MI instruction. The user can specify selection criteria according to the results of the group-by operation. Therefore, only those results that interest the user will be returned (for example, return the maximum salary of different groups of employees only when the maximum is greater than \$200,000). The group-by selection criteria is specified by a selection template to the Create Cursor instruction.

Although the data base supports security only on the object level, differences in the authorizations required to use the Create Cursor and Activate Cursor instructions allow the implementation of field level security through the use of a subset of the fields and/or different in- and out-maps.

Activating a Cursor

A cursor is activated by the Activate Cursor instruction. Activating the cursor binds the cursor, data spaces, and data space index (if used) to the process.

The activated cursor can be used only by the activating process, but the data spaces and data space index can be shared among multiple processes. The activated cursor, in addition to controlling the mapping to and from the user's view, also contains process-dependent information. The activated cursor contains the current position of the cursor (which is the entry to be retrieved) and the starting position to use in order to locate the next or previous entry.

Data space entries can be locked to a cursor and, therefore, to a process in order to control their access between the time their values are retrieved and the time they are subsequently updated or deleted.

Inserting an Entry

A new entry is inserted into the data space via an activated cursor and the Insert Data Space Entry instruction. The system determines the location for the new entry and then maps the data into the data space according to the field descriptions in the data space and the user's view of the data as defined in the cursor. The user selects the data space into which the entry is inserted by specifying that data space in the option list of the Insert Data Space Entry instruction or the Insert Sequential Data Space Entries instruction. The Insert Data Space Entry instruction inserts a single data space entry into the specified data space. The Insert Sequential Data Space Entries instruction inserts as many entries as requested into the specified data space. When the entries are inserted, all valid data space indexes over the data space are updated to reflect these entries. Inserting an entry does not require or change the positioning of the cursor.

Finding an Entry without a Data Space Index

To retrieve, update, or delete any data space entry or joined entry, the cursor must first be positioned to that entry. An activated cursor is positioned to an entry by the Set Cursor instruction. If the entry is to be updated or deleted, the user also specifies that the entry is to be locked.

The options available in positioning a cursor (without a data space index) are: first, last, next, previous, same, relative, and ordinal. The cursor can be positioned to the first or last entry in any data space. The next or previous entry can be found from the current position of the cursor. The relative option allows positioning of the cursor relative to the current position of the cursor (for example, here-plus-ten or here-minus-four). Ordinal positioning specifies a specific entry (for example, forty-third entry).

Retrieving an Entry

Retrieving a Single Entry

The Retrieve Data Space Entry instruction retrieves the entry indicated by the current position of the cursor. The system retrieves the entry from the data space and maps it into the program's storage area according to the field descriptions in the data space and the user's view of the entry as defined in the cursor.

Retrieving Multiple Entries

The Retrieve Sequential Data Space Entries instruction provides the functions of multiple set cursor and retrieve data space entry operations. During the retrieve operation, the system sets the cursor to the adjacent entry in the data space and then maps the entry into the program's storage area.

Retrieve Sequential allows the positioning and retrieval of data in a next or previous direction.

Updating an Entry

The Update Data Space Entry instruction causes an update of the entry that has been locked to the cursor for the longest time. In order for an entry to be updated, the entry must first be locked via the Set Cursor instruction. Then the entry can be retrieved via the Retrieve Data Space Entry instruction. The user processes the retrieved data (if any) and modifies it in the user program. The user then requests, via the Update Data Space Entry instruction, that the replacement of the updated data be moved back into the data space. The system maps the data back into the original entry in the data space according to the field descriptions in the data space and the user's input view of the entry (as defined by the input mapping definitions associated with the cursor. If the update causes a change in the value of a key field, all valid data space indexes that refer to the data space are updated to reflect the change. The entry is then unlocked from the cursor.

Deleting an Entry

The Delete Data Space Entry instruction selects the data space entry that has been locked to the cursor for the longest time and deletes it from the data space in which it resides. An entry is deleted from a data space in a two-step operation: the user must lock the desired entry in the data space via the Set Cursor instruction; the user then requests the deletion of the entry from the data space via the Delete Data Space Entry instruction. All valid data space indexes over the data space are updated to reflect the deletion of the entry. (The entry cannot be retrieved after deletion.) The entry is then unlocked from the cursor. A deleted entry leaves a void in the data space that can only be reused through a special option on the Set Cursor and Update Data Space Entry instructions.

De-activating a Cursor

A cursor can be explicitly de-activated through use of the De-activate Cursor instruction or implicitly through use of the Destroy Cursor instruction. De-activating a cursor causes all entries locked to the cursor to be unlocked and the cursor, data spaces, and data space index to be detached from the process. De-activating a cursor ensures that all inserts, updates, and deletes are reflected in auxiliary storage.

Destroying a Cursor

A cursor is removed from the system through use of the Destroy Cursor instruction. If the cursor is not active, the cursor is destroyed. If the cursor is active to the current process, the cursor is first de-activated and then destroyed.

Destroying a Data Space

A data space is removed from the system through use of the Destroy Data Space instruction. No active cursors or data space indexes can be over the data space when the Destroy Data Space instruction is executed.

Creating a Data Space Index

A data space index is created by providing an index template and issuing a Create Data Space Index instruction. The index template contains a complete description of the index to be built, points to all data spaces covered by this index, and defines a composite key for each data space. A composite key is made up of fields from the data space entry and, optionally, single-character constants called fork characters.

The Data Base Management user can optionally specify to create the index from an existing data space index. A data space subset under the existing data space index is specified as a subset to be associated with the data space index the user will create.

The data space index provides a logical ordering for entries in the data spaces by defining a key for each entry from each data space and then ordering the keys based on their values. Ordering attributes such as ascending or descending sequence, internal form, algebraic or absolute value, zone or digit force, and alternate collating sequence may be specified for each field from the entry in the key. The only difference between the keys for different entries from the same data space are the values of the fields in the entry that are used in the key.

The user can optionally define derived mapping operations to be performed on the fields before building the keys from them.

The fork characters and ordering attributes are the same for every entry within a data space. But they need not be the same for different data spaces or for different views of the same data space in a data space index. After the keys are built according to the field attributes and fork characters, the keys for all entries in all data spaces addressed by the data space index are ordered in ascending sequence. This sequence defines the logical ordering of the entries as seen through the data space index.

A data space index does not have to contain a key for every data space entry under it. A selection routine can be supplied in order to determine whether or not a key should be included. This optional routine (supplied when the data space index is created) is permanently bound to the data space index. The selection criteria may also be supplied in the form of a Selection Template without providing a separate routine.

Each time an additional entry is inserted into the data space or an entry is modified within a data space, selection determines, based on the values of fields within the entry, whether the index is to address the entry.

Finding an Entry with a Data Space Index

When a data space index is used to retrieve, update, or delete any entry, the cursor must first be positioned to that entry through use of the Set Cursor instruction. This is necessary because the positioning options are quite different from those without a data space index. The possible options are: first, last, next, previous, next unique, previous unique, next equal, previous equal, same, relative, ordinal, and five kinds of keyed operation. The relative and ordinal options function just as they do without an index.

The first, last, next, and previous operations find the appropriate entries with respect to their logical order as designated by the index rather than with respect to their physical order. The next unique and previous unique operations find the next or previous entry whose key differs from the key associated with the currently addressed entry. The next equal and previous equal operations find the next or previous entry whose key is equal to the key associated with the currently addressed entry. The key operations work with a user-provided key and find the entry whose key is before, equal or before, equal, equal or after, or after the specified key. All other functions, such as the locking of an entry, work the same with or without a data space index.

Retrieving Multiple Entries with a Data Space Index

The Retrieve Sequential Data Space Entries instruction can be used even though the cursor addresses data space entries through a data space index. Multiple set cursor and retrieve data space entry operations are performed in the logical order designated by the data space index. Data space entries are not locked for update through the Retrieve Sequential Data Space Entries instruction.

Destroying a Data Space Index

A data space index is removed from the system through use of the Destroy Data Space Index instruction.

Copying Data Space Entries

The Copy Data Space Entries instruction allows the user to copy data space entries. Entries in one data space can be copied to another data space. In addition, entries can also be copied from a data space back to the same data space. This instruction allows various options such as:

- Remove deleted entries
- Limit the number of entries to be copied
- Locate the entries to be copied
- Copy the entries in the logical order provided by a data space index or in the physical order of the entries in the data space

Shared Data Spaces

Data spaces and data space indexes are objects that can be shared. Multiple user processes can access a shared data space or a data space index concurrently. There are six lock states that a data space can assume:

Lock State	Other Processes Can Have
1 No lock	1, 2, 3, 4, 5, 6
2 Shared read (LSRD)	1, 2, 3, 4, 5
3 Shared read only (LSRO)	1, 2, 3
4 Shared update (LSUP)	1, 2, 4
5 Exclusive-allow read (LEAR)	1, 2
6 Exclusive no read (LENR)	1 only

The Lock Object instruction is used to lock a data space to a process. This instruction can be issued before or after the Activate Cursor instruction. (A data space can be used even though it is not explicitly locked to a process.) The activation of a cursor applies an implicit shared read lock on the data space index, and an implicit lock on the data spaces with the level identified in the activate cursor template; the default is a shared read lock. This prevents data space indexes and data spaces from being destroyed by other processes.

Lock states 1 and 2 (no lock and shared read) permit multiple processes to concurrently update the same data space. This means that some further level of lock protection must exist at the entry level. Data base management, therefore, locks the entry to the cursor between the time the cursor is positioned and the time the entry is updated or deleted. This guarantees that only one process can be changing the entry. Figure 5-1 is an example of a shared data space.

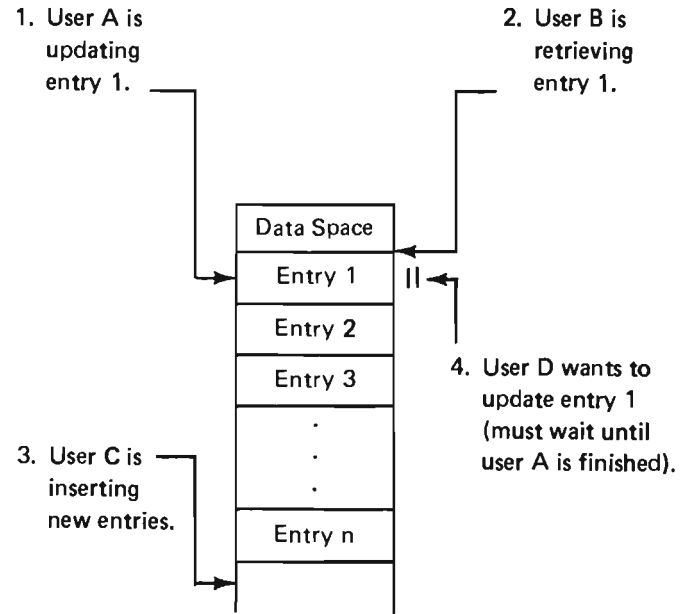


Figure 5-1. A Shared Data Space

The Set Cursor (for update) instruction applies an implicit shared update lock on the data space (if the Activate Cursor instruction applied a shared read lock or a shared read-only lock on the data space), and an implicit exclusive-allow read lock on the data space entry. This lock arrangement prevents other users from attempting to concurrently update the data space entry; but the locked entry can be retrieved by a different process if its Set Cursor instruction does not specify that the entry be locked. While an entry is being updated, data base management prevents any other access to the entry to ensure that a partially updated entry cannot be retrieved by a different process.

Multiple Locked Entries

Multiple entries can be locked before they are updated. This function is useful in a shared environment because a program can guarantee that all entries to be updated are actually available (not held by another program) before starting to update them. The lock on the entry is applied when the entry is addressed via a Set Cursor instruction that specifies the lock for update option. Addressability to the entry is put in a queue that identifies all entries that have been locked through the cursor. The entry can be added to the beginning or the end of the queue. When an Update Data Space Entry instruction or a Delete Data Space Entry instruction is issued, addressability to the first entry referenced via the queue (the entry that is locked at the beginning of the queue) is removed and then used to address the entry to be modified.

If an operation consists of five entries to be modified, issuing a Set Cursor instruction for each entry locks all the entries before any changes are made. The entries can then be changed in the data space by issuing five successive Update Data Space Entry instructions or five successive Delete Data Space Entry instructions. This procedure automatically clears addressability to the entries from the queue in FIFO order.

A potential deadlock situation exists anytime a process sequentially locks more than one resource. Deadlocks can occur when locking data space entries because data space entries are considered to be a resource (from the standpoint of locks). Resource management provides the basis of a deadlock prevention mechanism by way of the lock wait time-out exceptions. When such an exception occurs, it can mean that a potential deadlock has occurred. Generally, the user should respond to a lock wait time-out exception by releasing all resources held by the process and then once again attempt the lock operations.

The user can constrain the order in which locks are acquired. This creates an environment where only locked data space entries need to be released and relocked when a time-out occurs while an attempt is being made to lock a new data space entry.

Unlike other resources in the machine, a single process cannot hold more than one lock at a time on a data space entry. It is possible, then, for a process to appear to deadlock itself by attempting to lock an entry that it has already locked. To assist in diagnosing this condition in the user program, data base management, when signaling the lock wait time-out exception, returns an indication that the entry is already locked to the current process.

Ensuring Changes

Ensuring changes is the process of guaranteeing that changes to the data base are in auxiliary storage as protection against a system failure. Data base management provides several methods of ensuring changes to the data base.

Ensuring One Entry at a Time: The Set Cursor instruction has a forced write option that is specified when a data space entry is locked. Specifying this option causes the corresponding update or delete operation to force the entry to auxiliary storage before completion of the instruction. This option is also available on the Insert Data Space Entry instruction.

Ensuring Multiple Inserts: The Insert Sequential Data Space Entries instruction has a forced write option that causes all of the entries inserted by the instruction to be forced to auxiliary storage before completion of the instruction.

Ensuring Multiple Entries: The Ensure Data Space Entries instruction causes data spaces that had entries changed through the specified cursor to be transferred to auxiliary storage.

Note: The Ensure Data Space Entries instruction does not necessarily ensure the data space indexes that reference the data space.

De-activate Cursor: The De-activate Cursor instruction performs the same function as the Ensure Data Space Entries instruction would perform on the same cursor.

Ensure Object: The Ensure Object instruction can be used at any time to ensure a data space or a data space index. If the object is a data space, it will be transferred to auxiliary storage. If the object is a data space index, it will be transferred to auxiliary storage along with all associated data spaces.

Data Space Index Maintenance

Data base management guarantees that every valid data space index correctly identifies the entries in the data spaces it addresses. To do this, data base management automatically maintains every valid data space index in the system.

To maintain this guarantee, data base management must invalidate a data space index any time there is any question as to its correctness. During execution time, an event is used to indicate to the user that the data space index was invalidated; the object recovery list is used for this same purpose during the recovery portion of an IPL (initial program load).

A unique keyed data space index might become invalid during execution time. If this occurs, data base management does not allow any inserts or updates to the data spaces under the invalid data space index. This restriction assures that the data space index can be rebuilt.

Note: This restriction does not apply to data spaces under explicitly (user-requested) invalidated data space indexes.

Indexes are rebuilt through the Data Base Maintenance instruction; if a user does not want to have a data space index maintained, he can either invalidate it through use of the Data Base Maintenance instruction, or initially create the data space index invalid so that it will not be continually maintained.

Data base management also supports a delayed maintenance option on data space indexes (except when the data space index requires unique composite keys). That is, if a data space index needs maintenance and no active cursor references that specific index, then maintenance is delayed until either a cursor that references the index is made active or until the index is explicitly rebuilt by the Data Base Maintenance instruction.

Materialization of Data Base Object Attributes and Statistics

A materialize instruction can be issued for each data base object. This allows the user to materialize either the creation template for the object or the current statistics for that object.

Modification of Data Base Object Attributes

A modification instruction can be issued for the Data Space and Data Space Index data base objects, which allows the user to modify certain creation and operational attributes for that object.

Recovery Considerations

Changes to Data Space Entries: Data base management guarantees sequentiality of inserts but does not guarantee the integrity of updates or deletes for data spaces active at the time of a system failure. Insert sequentiality means that, after recovery, if an inserted entry is in the data space, every previously inserted entry in that data space is also present.

Data Space Entry Locks: All data space entries that are locked when a system failure occurs are automatically unlocked when an IPL is performed.

Implicit Locks: All implicit locks on data base objects are automatically removed during an IPL.

Data Space Indexes: A data space index, which contains addressability to an active data space during a system failure, may be invalidated or damaged during an IPL. The invalidated or damaged data space index is listed in the object recovery list during IPL.

Active Cursors: A cursor that is active during a system failure will appear to be de-activated following an IPL and will be recovered and activated when the next Activate Cursor instruction is issued to that cursor.

Temporary Objects: All temporary objects are destroyed when an IPL is performed.

Creates: All data base objects being created at the time of system failure are destroyed when an IPL is performed.

Data Space Indexes Being Rebuilt: Any data space index being rebuilt at the time of the system failure is invalidated and listed in the object recovery list when an IPL is performed.

Data Space Being Reorganized: When a system failure occurs while a data space is being reorganized through the Copy Data Space Entries instruction, that data space will be either unchanged or completely reorganized when an IPL is performed. Data space indexes over the data space might be invalidated during the IPL.

Data Spaces Being Copied To: Any data space that is the receiver for a Copy Data Space Entries instruction during a system failure, might contain all the copied entries, some of the copied entries, or none of the copied entries after an IPL is performed. If some of the copied entries are contained in the receiving data space, the sequentiality of inserts applies.

Data Spaces Being Journalled: Any data space that is being journalled will be synchronized with the journal during an IMPL (initial microprogram load). All changes for the data space that have been entered in the journal space are then reflected in the data space.

User Information after IPL: After recovery, the following data base information can be materialized by the Materialize Machine Attributes instruction:

- Objects detected as damaged during IPL (data spaces, data space indexes, and permanent cursors).
- Data spaces active during a system failure.
- Data space indexes that are invalidated when an IPL is performed.
- Any activity that was performed as a result of synchronizing a data space with the journal.
- Any failure to synchronize the data space with the journal.

Performance Considerations

The following considerations may improve performance of the data base:

Overlapped Data Base Operations: A program can process current entries in main storage while fetching additional entries from auxiliary storage.

Unit of Transfer: When a cursor is created or activated, the user can specify a unit of transfer. This is used when the entries not in main storage are to be processed sequentially. It means that when an entry is fetched, the adjacent n-1 entries are fetched at the same time because they are also likely to be used.

Processing Mode: When a cursor is created or activated, processing mode can be specified to indicate whether the keys in the data space index and/or the actual data space entries are to be processed randomly or sequentially. Processing mode is used by data base management as an aid in bringing and purging pages to improve data base paging characteristics.

Unit Specification: When a data space or a data space index is created, a preferred unit can be specified. Data base management attempts to keep the object on the preferred unit and indicates, as a return value in both the create and materialize creation template options, whether or not the object is all on the preferred unit. This allows the user to place different data base objects on separate units in order to control disk loading.

Contiguous Return Bit: When a data space is created, the user can optionally specify the allocation of a contiguous space for the data area. At this time or when the creation template of the data space is materialized, a contiguous return bit is available to indicate whether or not the data area of the data space is contiguous. If the data area is contiguous, performance in sequential accessing situations may be improved.

Direct Map: Even though mapping and conversion are optimized as much as possible, a view that is identical to the physical format of the entry requires less processor time than a complicated user's view of the entry.

Retrieve Sequential Data Space Entries: This instruction may provide better machine performance than multiple Set Cursor and Retrieve Data Space Entry instructions under any one of the following conditions:

- Numerous entries are to be retrieved from the data space by the process.
- The number of entries to be retrieved for each instruction is not excessive relative to the process storage pool size.
- The entries are to be retrieved in either increasing or decreasing ordinal number sequence.

This instruction may provide better machine performance than a combination of one Set Cursor and one Retrieve Data Space Entry instruction under the following conditions:

- The entries are to be retrieved in either increasing or decreasing ordinal number sequence.
- The amount of processing for each entry is minimal.

Insert Sequential Data Space Entries: This instruction may provide better machine performance than multiple Insert Data Space Entry instructions under any one of the following conditions:

- The size of the interface buffer and the number of entries to be inserted match the storage pool size.
- Numerous entries are to be inserted into the data space by the process.
- Duplicate key conditions are not expected.

Optimizing Data Space Index Usage: When a data space index is created, the user can specify that the index is to be used for either random or sequential retrieval of entries through a cursor over the index. Improved machine performance may result if the processing of the data space entries through the index is predominately of the specified type.

Delayed Maintenance: When delayed maintenance is specified for a data space index, improved machine performance may result when an insert, delete, or update operation is performed on the data space entries under that index. However, the creation or activation of a cursor over this data space index may require additional time because the maintenance of the index is performed at this time. This option is not available when the index requires unique composite keys.

Journalled Data Spaces: Journal operations (such as making the journal entries that are related to data space changes) can cause degradation of machine performance.

If machine performance is decreased, the user should determine if the increased processor time is worth the advantages provided by journal management.

The user should also review the data base management plan to determine if certain operations (such as Ensure Data Space) are still being utilized even though they are no longer needed because of journal operations.

Data Base Maintenance Functions

The Data Base Maintenance instruction provides six general functions to aid in maintaining data spaces and data space indexes.

Rebuild Data Space Index: This function rebuilds an invalid data space index from the data spaces under it. Automatic or delayed index maintenance is resumed.

Invalidate Data Space Index: This function invalidates a data space index. This causes automatic or delayed index maintenance to stop and the index to be unusable. The index must be either rebuilt or destroyed. Invalidating a data space index frees system space for other uses and also reduces the amount of time required for index maintenance.

Reset Data Space: This function causes the data space to be reset so that it no longer contains any entries.

Reset Data Space with Number of Entries Specified: This function causes the data space to be reset so that the data space does not contain any entries. In addition, space is allocated for the specified number of entries.

Increment Maximum Number of Entries: This function causes the user-specified maximum number of entries for a data space to be incremented by the amount specified by the user.

Insert Deleted Entries: This function adds a user-specified number of deleted entries to the end of the data space.

Insert Default Entries: This function adds a user-specified number of entries to the end of the data space. Each entry contains the default values for all fields in the entry.

DATA SPACES

A data space is a system object in which entries of data reside. An entry is composed of an ordered set of fields with each field having an associated set of attributes. (The field is the smallest unit that is recognized by the data base.) All entries in a data space are homogeneous; that is, each entry has the same field attributes as all other entries in that data space. A program can insert a new entry into a data space; or a program can update, retrieve, or delete an existing entry.

The Create Data Space instruction is used to create a data space. This instruction provides a description of all the fields contained in an entry. This description remains constant for the life of the data space and describes how the data is actually stored in the data base.

When a cursor is created to use a data space, the user must provide another description called the mapping template. The mapping template describes how the user wants the entries to appear to the user program. The mapping template is the vehicle used for achieving late binding, multiple views, and data independence. A mapping template allows the user to assign different attributes to fields, rearrange the perceived order of the fields in an entry, and extract a defined subset of fields from an entry for use in the using program. The user may provide two mapping templates when the transformation of data is different for retrieval than for insert or update. Thus, the program may be able to retrieve fields it cannot update and/or insert. Conversely, the program may be able to update and/or insert fields it cannot retrieve.

Figure 5-2 shows a data space and the associated attributes of each of the fields in the data space. This figure also shows a view of the data space as seen through a mapping template. Notice that the mapping template view of the data is different from the actual data in the data space. The user's view contains only three fields that appear in a rearranged order and have different attributes than those actually stored in the data space.

The user's view of entry 2 differs from the actual data in the data space:

- In the user's view, only three fields are mapped for retrieval.
- The attributes of field B are represented differently.
- The three fields in the view are in a different order.

User's View

Fields	A	D	B
Attributes	Character (10)	Decimal (5.2)	Decimal (5)
Values	Jack Jones	123.45	00032

Data Space

Fields	A	B	C	D	E
Attributes	Character (10)	Binary (2)	Binary (4)	Decimal (5.2)	Character (5)
Entry 1					
Entry 2	Jack Jones	Hex 0020	Hex 00000002	123.45	ABCDE
3					
4					
.					
.					
.					
.					
Entry n					

Figure 5-2. Data Space and Associated Attributes

Data Space Organization

Entries in a data space are stored according to arrival sequence. (The $n+1$ entry is stored after the n th entry in the data space.) A data space can be accessed directly or indirectly (through use of a data space index). When accessed directly, the relative positioning and ordinal positioning types of addressing are allowed.

Relative positioning means that the address of an entry is based on the address of the currently addressed entry. Relative positioning can be either forward or backward. Fetching the previous entry, the next entry, or the fourth entry after the current entry are examples of relative positioning.

Ordinal positioning means addressing an entry by its absolute entry number in the data space. Ordinal positioning can only be forward (positive) from the start of the data space. Fetching the tenth entry is an example of ordinal positioning.

A data space can be created with a limit on the number of entries to be allowed in the data space. The presence of this limit can be used to prevent runaway situations. For example, when the Insert Data Space Entries or Insert Sequential Data Space Entries instruction attempts to exceed the limit, an exception occurs. The Data Base Maintenance instruction can be executed (with the increment maximum number of entries option specified) to cause the maximum number of entries to be incremented by the specified amount. The instruction that caused the exception can be reissued; the entry or entries are then inserted.

DATA SPACE INDEXES

A data space index is a system object that is used to logically reorder the entries in one or more data spaces. A data space index accesses the data space independently of the physical ordering of the entries. Multiple data space indexes can provide different logical orderings of the same data. Data space indexes provide a wide range of functions needed to cover differing applications. Some of these functions are:

- Addressing by key
- Addressing by generic key
- Relative positioning
- Differing-length composite keys
- Ascending versus descending order with respect to the key field value
- Signed and unsigned numeric key fields
- Ordering duplicate keys (between data spaces and within a data space)
- Alternate collating sequence on any key field
- Index addressability to subsets of entries within data spaces, as defined by the user's selection criteria
- Automatic index maintenance

Types of Addressing for Data Space Indexes

The following types of addressing are allowed with a data space index:

Key: An entry can be accessed on the basis of its composite key value.

Generic Key: An entry can be accessed on the basis of a leading portion of the composite key value.

Sequential: The next or previous entry addressed by the data space index can be accessed.

Sequential Unequal: The next or previous entry with a key value not equal to the key value of the current entry can be accessed.

Sequential Equal: The next or previous entry with a key value equal to the key value of the current entry can be accessed.

Approximate Key: If the exact search argument does not exist in the data space, the entry with the next higher or lower key value than the key value used as the search argument is accessed.

Data Space Index Keys

Data space index keys are composed of one or more fields from the data space entry. The key for one data space under a data space index is not required to have a format identical to the composite key for another data space under the same data space index. In other words, variable-length keys can exist between data spaces for the same data space index.

The user can specify logical keys. Normally, the key fields are taken from the contents of the specified data space entry field. With logical keys, derived field operations can be performed on the key field. The resulting field is used as the key field in the index (for example, key field = area code field concatenated to local telephone number field).

The key fields that comprise the composite key can specify certain attributes to force an ordering. Each key field can have the following modification attributes:

- Arrange in ascending order—No change is made to the field.
- Arrange in descending order—The field is modified (in the index) so that the values appear in reverse order in the index.
- Algebraic—Valid only for numeric fields. The machine does whatever is necessary for each numeric type to force the values to be ordered from minus infinity to zero to plus infinity. (Without this, different internal representations would be ordered differently; for example binary numbers would be ordered 0,1, . . . ,+infinity,-infinity, . . . ,-1.)
- Absolute value—Valid only for numeric fields. The machine does whatever is necessary for each numeric type to ensure that the values are ordered in absolute value order.
- Internal form—Neither absolute nor algebraic.
- Alternate collating sequence—Valid only for character and zoned decimal fields. The machine causes a character or zoned decimal field to be translated by the user-supplied alternate collating template.
- Zone or digit force—Valid only for character or zoned decimal fields. The machine causes the zone or digit bits of every byte in a character or zoned decimal field (in the index) to be forced to zeros.

The same key field cannot be arranged in both ascending and descending order. Neither can an algebraic and an absolute value be specified for the same field. However, other combinations are valid. For example, if algebraic and descending order are specified for a numeric field, it would be ordered: +infinity, . . . ,+1,0,-1, . . . ,-infinity.

A forced ordering of keys between data spaces can be achieved through the use of fork characters. A fork character is a single character constant that is defined for a specific position in the key of a data space. The fork character is not contained in the data space entry itself, but is inserted into the key for that data space entry when the key is built by the system. The fork character is the user's way of controlling the ordering of the index between data spaces. For example, with an order header file and an order item file, fork characters can be used to cause the order header entry to appear before the order item entries for that order when accessed through the data space index sequentially. To do this, the following key templates can be defined:

Order Header File Key

Field	Fork Character
ORDER NUMBER	0

Order Item File Key

Field	Fork Character
ORDER NUMBER	1

When the entries are accessed sequentially through the index, the entries appear in the following order:

Key	Entry
325 0	ACE HARDWARE, BROOKLYN, NY
325 1	SCREWS 325-106795
325 1	NAILS 600-239479
326 0	JONES HARDWARE, AMES, IOWA
326 1	NAILS 1800-239479
.	.
.	.
.	.

Within a data space, the ordering of two or more entries with duplicate keys is determined by the specification of one of the following duplicate key rules.

- LIFO (last-in-first-out)—The entry with the highest ordinal (last inserted) is placed first in the data space index.
- FIFO (first-in-first-out)—The entry with the lowest ordinal (earliest inserted) is placed first in the data space index.
- Unique—Duplicate keys are not allowed. There cannot be a duplicate key of the same defined length anywhere in the index.

Index Addressability to Subsets of Data Space Entries

Normally, a data space index contains a key for each entry in each of the data spaces to be addressed by that data space index. The user can, however, construct a data space index that does not contain keys for all of the entries in the data space(s), but only for a selected subset of the entries. To accomplish this, the user must provide a selection routine that determines whether or not an entry is to be addressed by the data space index. Or the user can elect to use a selection template to determine if an entry should be addressed by the data space index. Either method can be used. The effect of using the selection template is the same as if the selection routine was used. A user cannot specify both of these when creating the index. The user must also provide a description of the data space entry fields that are to be passed to the selection routine for each data space.

Each time the index is updated, the machine calls the selection routine and the identified fields mapped from the data space entry are passed to the selection routine. The selection routine makes a yes or no decision and then, if the decision is yes, the index maintenance mechanism updates the data space index.

This allows a data space index to be used as an access path to any subset of the data in the underlying data spaces. Because the selection criteria are user-determined and user-controlled, there is virtually no limit to the type of selection criteria that can be applied to an entry except that the decision must be based entirely on the data in that entry.

Example of Data Space Index Ordering

The following shows the effect of different key characteristics on the ordering of a data space index. Assume that a data space index is to be built over the three data spaces shown in Figure 5-3. The data spaces represent the following information:

- Data space 1 contains employee master entries that are indexed by Employee Number.
- Data space 2 contains employee job history entries that are indexed by Employee Number and Code.
- Data space 3 contains employee education entries that are indexed by Employee Number and Date.

The intent is to construct the index so that the three data spaces appear as a hierarchical structure in which the machine can retrieve an employee master entry, followed by all of the job history entries, and then by the education entries for that employee.

**Data Space 1
Employee Master**

Employee Number	Data
122	
123	
.	
.	
.	

**Data Space 2
Employee Job History**

Employee Number	Code	Data
122	02	
123	02	
122	04	
123	05	
122	06	

**Data Space 3
Employee Education**

Employee Number	Date	Data
122	011271	
123	010572	
122	022072	
122	030173	
123	040573	

The entries are to appear as follows:

```

Employee Master
Job History
.
.
.
Job History
Educational History
.
.
.
Educational History
Employee Master
.
.
.

```

This ordering can be achieved by creating an index and specifying the appropriate fork characters between the fields of the composite keys as follows:

- For data space 1 (employee master), the key is:
Employee Number | 0 |
- For data space 2 (employee job history), the key is:
Employee Number | 1 | Code
- For data space 3 (employee education), the key is:
Employee Number | 2 | Date

Figure 5-3. Data Space Examples for Index Ordering

After the data space has been created, the entries will be ordered by the index as follows:

	Portions of the Key Supplied by the User			Portions of the Key Appended by the Machine	
	Employee Number	Fork Character	Code or Date	Data Space Number	Ordinal Entry Number
Data Space Index Keys	122	0		1	1
	122	1	02	2	1
	122	1	04	2	3
	122	1	06	2	5
	122	2	011271	3	1
	122	2	022072	3	3
	122	2	030173	3	4
	123	0		1	2
	123	1	02	2	2
	123	1	05	2	4
	123	2	010572	3	2
	123	2	040573	3	5

Notice that each data space has a different key format; however, this does not mean that the formats could not have been the same. The key format for each data space might have been identical (for example, Employee Number), and the fork characters might not have been used. In that case, the ordering of duplicate keys (Employee Number) would have been by data space and within a data space would have been ordered by arrival sequence.

Notice that the key fields are in ascending order. If the Employee Number key field had a descending order attribute specified, the keys with Employee Number 123 would appear before the keys with Employee Number 122. The ordering among the keys with Employee Number 123 would not change, however, because the attribute only affected the Employee Number field. The same is true of the Code field. If it had a descending order attribute, the three keys with Code 02, 04, and 06 would be reversed as would the two keys with Code 02 and 05. The rest of the ordering would be unaffected.

The fork characters utilized in this example exhibit significant control over the order among entries containing the same employee number. The employee master entries come first, followed by all of the job history entries for that employee, followed by all of the educational history entries for that employee. Without the use of the fork characters, it would not be possible to maintain this degree of control over the ordering of the index with respect to a hierarchy of record types.

Data base management automatically inserts the user-defined fork characters into the keys when updating or searching the data space index. There can be multiple fork characters within any one key, and they do not have to be between every field. This permits having many different levels of fork characters and thus many levels of a hierarchical structure.

The user must determine what fork characters are needed and where they are to be located in the composite keys. Notice the 0 fork character on the end of the key for data space 1; the data base appends internal information to the end of a key in creating the key. Without the fork character, the appended information will interfere with the ordering. The 0 fork character is not needed if there are no key definitions that are longer, as in data spaces 2 and 3.

Notice that the data space index has no duplicate entries. Duplicate entries can be inhibited by creating the data space index with the unique attribute. For example, duplicate keys may be generated during the execution of an Insert Data Space Entry or an Update Data Space Entry instruction; but if this occurs while the machine is updating any of the data space indexes marked unique, the insert or update is rejected and the data space remains unchanged.

If the data space index allows duplicate keys and if the ordering is FIFO, the duplicate keys appear in the same order as they do in the data spaces. Entries for data space 1 are first according to their order in the data space, followed by entries for data space 2. If LIFO was specified, the entries for data space 1 would still appear before the entries for data space 2, but duplicates within a data space would appear in the opposite order.


Because the key formats might differ between the composite keys for multiple data spaces, the user must specify which data space to use when the composite key is generated for retrieval purposes. This is necessary so that the data base can determine which fork characters to insert into the key and what ordering modifications to make on the key fields before searching the data space index. It is not necessary to specify the data space when retrieving sequentially via the data space index. Thus, in the preceding example, to retrieve all of the entries for employee 123, the user must specify that the entry from data space 1 is desired and the key is 123. After retrieving this entry, the user simply gets the next entry until the value for the Employee Number field changes.



Journal Management

Journal management provides a journal (record) of the current changes made to selected system objects. The journal can then be used as an activity trail or be used during a recovery operation.

Journal management:

- Records changes to those objects that are having their changes journaled. The information supplied with each journal entry indicates when the change was made, which process made the change, what user caused the change, and the user program that was executing when the change was made.
 - Performs various operations (during IMPL) to correct any mismatch between the changes on the journal space and the changes to the appropriate objects.
 - Allows the machine interface user to place entries in the journal space.
 - Provides a variety of search criteria for retrieving entries from the journal space.
 - Recovers an object by using the changes entered in the journal space.
 - Provides enhanced recovery capabilities.
- 

JOURNAL OBJECTS

The journal port and journal space system objects are used by the journal management functions.

Journal Port

A journal port provides the mechanism for linking those objects specified as journaled objects to journal spaces. A journal port also provides a definition of the prefix data associated with each entry in the journal space. Additional prefix data can come from the object being changed, the user's MI program, the process that caused the change, and the user profile controlling the execution of the process.

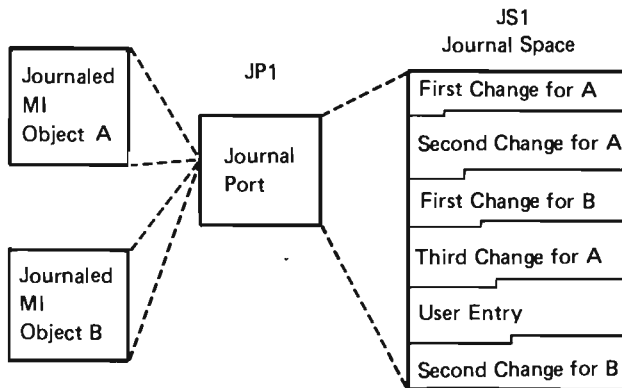
Journal Space

A journal space can receive entries after it is attached to a journal port. Thereafter, all changes being journaled through the port are entered in the journal space.

Either one or two journal spaces can be attached to a journal port. When two journal spaces are attached, they must be attached at the same time. During journal operations their contents will be identical unless a failure occurs. If a failure occurs while the system is adding entries to one of the journal spaces, and the system can place the entry on the second journal space, no exception is signaled. Journal management continues to make entries only in the undamaged journal space.

Once a journal space is detached from a journal port, it cannot be attached to a journal port again. However, the search function as well as the recovery procedures can still be executed against the detached journal space.

The following illustration shows MI (machine interface) objects A and B being journaled through journal port JP1. Journal space JS1 is attached to journal port JP1 and, therefore, is being used to record change entries for objects A and B. Each entry is sequentially recorded in the journal space. Serialization of operations on all the objects is maintained in the journal space entries regardless of the operation performed, the process performing the operation, or the object being operated on.



SPECIFYING OBJECTS TO BE JOURNALED

The Journal Object instruction causes the machine to start or stop journaling changes to a journaled object.

If a journal port and an object that can be journaled are specified on this instruction, a link is established between the object to be journaled and the specified journal port. Once the link is established, changes to the object are journaled through the port until either the object being journaled is destroyed or the Journal Object instruction is reissued without the journal port being specified.

An object can be specified as a journaled object an unlimited number of times but there can be only one link between the object and a port. An object cannot be journaled through more than one port at a time.

This instruction also allows the user to specify whether certain optional changes are to be made to the journal object.

JOURNAL ENTRIES

Single or multiple entries are created and placed in the journal spaces through a journal port for each change made to those objects whose changes are being journaled. (The journal spaces must be attached to a journal port.) The entries are not placed in the journal space until all checking has been performed and the journal operation is not completed if an error is detected. Once the entry is placed in the journal space, the operation will be completed.

The Retrieve Journal Entries instruction can be used to view selected entries in the journal spaces. Each entry in the journal space contains the following information.

Journal Entry Length	Journal Sequence Number	Entry Type	Entry Subtype	Status	Journal Prefix	Journal ID	Entry Specific Data
----------------------	-------------------------	------------	---------------	--------	----------------	------------	---------------------

Journal Entry in Journal Space

- The *length of the entry*, which designates the entire length of the journal entry, including the entry length field itself.
 - The *sequence number*, which is increased by one for each entry inserted into the journal space. It is set to one when the reset sequence number option is used on the Modify Journal Port instruction.
 - The *type of entry*, which for system objects is the object type. For example, this 1-byte value is a hex 0B when the entry is related to a data space system object. User entries are the hex 00 type; entries concerning the journal are the hex 09 type (journal port).
 - The *entry subtype*, which specifies the exact type of entry. For user entries this 2-byte value is specified by the user. For entries inserted by the machine, the subtype defines the format (but not the length) of the entry data. For example, the start journaling object value, which is inserted by the machine, is hex 0010.
 - The *status*, which specifies the journal ID option for this entry.
 - The *journal prefix*, which is made up of the following optional items and whose length is specified by the Create Journal Port instruction:
 - The *timestamp*, which represents the time that the entry was placed in the journal space.
 - The *process name*, which is the name of the process control space that is making the change.
 - The *user profile name*, which is the name of the user profile under which the process making the change is executing.
 - The *program name*, which is the name of the program that made the change.
 - The *journal ID*, which identifies the object to which the change was made. The ID is chosen by the user and is associated with the object at the time journaling is started. It stays with the object even after journaling has stopped.
- User entries do not require a journal ID. However, as a user, you may want to specify an ID in order to record a change you made to an object (for example, an important status change to an object's associated space).
- The *entry data*, which is defined by the type and subtype of the entry. Its length can be determined by subtracting the prefix length from the entry length.

APPLYING JOURNALED CHANGES

The Apply Journalled Changes instruction provides the user with a facility to reapply changes to or delete changes from an object. One use of this instruction is recovering objects.

Object Recovery Methods

One method of recovery consists of returning the object to a previous level (load/dump, copy, create) and then updating the object by applying the changes recorded in the journal space for that object.

Another method of recovery returns the object to some previous level. This is accomplished by applying to the object the before images of the changes recorded in the journal space. (A before image is a copy of the data in an object before a change is made.) The before images object journal attribute must be specified on the Journal Object instruction before this method of recovery is used.

The Apply Journalled Changes instruction is terminated if an error occurs when the machine is trying to apply changes to an object. The machine tries to return enough information to the user in order to restart the operation.

JOURNAL STATUS DURING IMPL

A system failure can cause the various storage facilities of the machine to be inconsistent with each other. During IMPL, journal management performs the following actions so that the entries in the journal space and the changes to objects are consistent:

- The last entry is found in the journal space and all header information in the journal space is updated to be consistent with that last entry.
- Any attached journal space that is damaged, partially damaged, unusable, or for which the threshold value has been exceeded is placed on the object recovery list. This list can be obtained through the Materialize Machine Attributes instruction.
- The journal port is corrected to match the journal spaces attached to it.
- If there are two journal spaces attached to the journal port, they are corrected so that their contents are identical.
- If the last operation specified on the journal was to either start recording the changes to an object or stop recording the changes to an object, then the operation is completed.
- An IMPL entry is placed on each journal port that has journal spaces attached.
- For those objects with journalled changes, the changes are applied from the last point that the journal and object were known to be consistent up to the end of the journal entries. This causes all journalled objects to be consistent both with the journal and with the other journalled objects.
- All journal ports are reported in the object recovery list.

In order to guarantee consistency between the journal and its associated objects, the machine must ensure that the journal entries reach nonvolatile storage before the associated object change. Journal management does not allow a journal space to be destroyed, suspended, or loaded over an existing journal space if it is needed for recovery.

LOAD/DUMP

A journal space can be dumped and loaded. A journal space can be dumped at any time. A journal space can be loaded over an existing journal space if it is empty, suspended, or if the dumped version has the same first sequence number and a last sequence number that is at least as large as the version in the machine.

COMMIT MANAGEMENT

Commit management, in conjunction with journal management, assists the MI user in maintaining the integrity of the data base management system.

Commit management allows the MI user to group a set of data base changes so that:

- The changes can be committed; that is, the changes made to an object under commitment control are made permanent and are ready for modification by the rest of the system.
- The changes that have not been committed can be decommitted; that is, the reverse image is applied (the changes are backed out) to an object whose changes were made under commitment control. Committed changes cannot be decommitted.
- If a process terminates without committing or decommitting the changes, they are implicitly decommitted.
- If a system failure occurs, any changes that have not been committed are implicitly decommitted during the subsequent IMPL.

The changes that can be committed or decommitted are insert, update, and delete data space entry.

COMMIT OBJECT

The commit block system object is used by commit management functions.

Commit Block

A commit block is a permanent system object that holds information concerning the changes made to objects under commitment control. The commit block contains a list of the objects under commitment control, a list of data space record lock identifiers, a list of objects that have undergone changes in the current commit cycle, and the commit description.

The following MI instructions are used to manage the commit block.

The Create Commit Block instruction is used to create a commit block.

The Destroy Commit Block instruction is used to destroy a commit block. A commit block cannot be destroyed if it is attached to a process.

The Modify Commit Block instruction is used to:

- Attach the commit block to a process. The commit block cannot be attached if it is already attached to any process. Whenever a commit block is attached to a process, only that process can modify it.
- Detach a commit block from a process. The commit block cannot be detached if a start commit entry has been placed in the journal but no commit or decommit operation has been performed. In addition, a commit block cannot be detached if there are any objects still under commitment control (objects still remain in the commit object list).

- Place objects under commitment control. One or more objects can be placed under commitment control.

The only objects that can be under commitment control are cursors. However, it is the changes to the data spaces under the cursor that can be committed or decommitted. The data spaces themselves then are not under commitment control. Another cursor, which is not under commitment control, can be over these same data spaces, and the changes made via this cursor are not under commitment control.

All objects changed under the control of the same commit block must have their changes journaled to a single journal port and that port must be the same port to which the commit block is journaled. For more information about journal objects, refer to *Journal Management* earlier in this chapter.

A cursor cannot be holding any data space entry locks when the cursor is placed under commitment control.

- Remove specific objects (cursors) from commitment control. The cursors cannot hold any data space entry locks.
- Remove all objects (cursors) from commitment control. The cursors must not hold any locks required by commitment control.

The Materialize Commit Block instruction returns either the creation template with current commit block attributes or the commit block status.

COMMIT DESCRIPTION

A commit description is a variable-length string of data that is specified on the Commit instruction. A commit description is not used by the machine because it contains only the data specified by the MI user. When the Commit instruction is executed, the commit description is placed in both the journal space and the commit block.

The Materialize Commit Block Attributes instruction can be used to materialize the commit description associated with the last successful execution of the Commit instruction.

COMMIT OPERATION

A commit operation can be performed only when the MI Commit instruction is executed.

The MI Commit instruction causes the changes to system objects, made under the control of a specific commit block, to be made permanent and available for modification by the rest of the system. The commit block must be attached to the process that issues the Commit instruction; otherwise, an exception is signaled.

The following occurs when a commit operation is performed:

- A start commit entry is placed in the journal space if a commit cycle has not been started.
- An entry, which contains the commit description, is placed in the journal space.
- The commit description is placed in the commit block.
- The data space index keys that were reserved to this commit block are removed.
- The data space entry locks that are held by the commit block are released.
- The data space entry locks that are held by all cursors under commitment control are released.
- The position of cursors under commitment control is not altered.
- The in-use count is decremented for any data space that had its count previously incremented by data base operations performed in the commit cycle prior to the current commit operation.
- The LSUP lock is released for any data space that had been previously locked by the data base operations performed in the commit cycle prior to the current commit cycle.

DECOMMIT OPERATION

A decommit operation can be explicitly or implicitly performed.

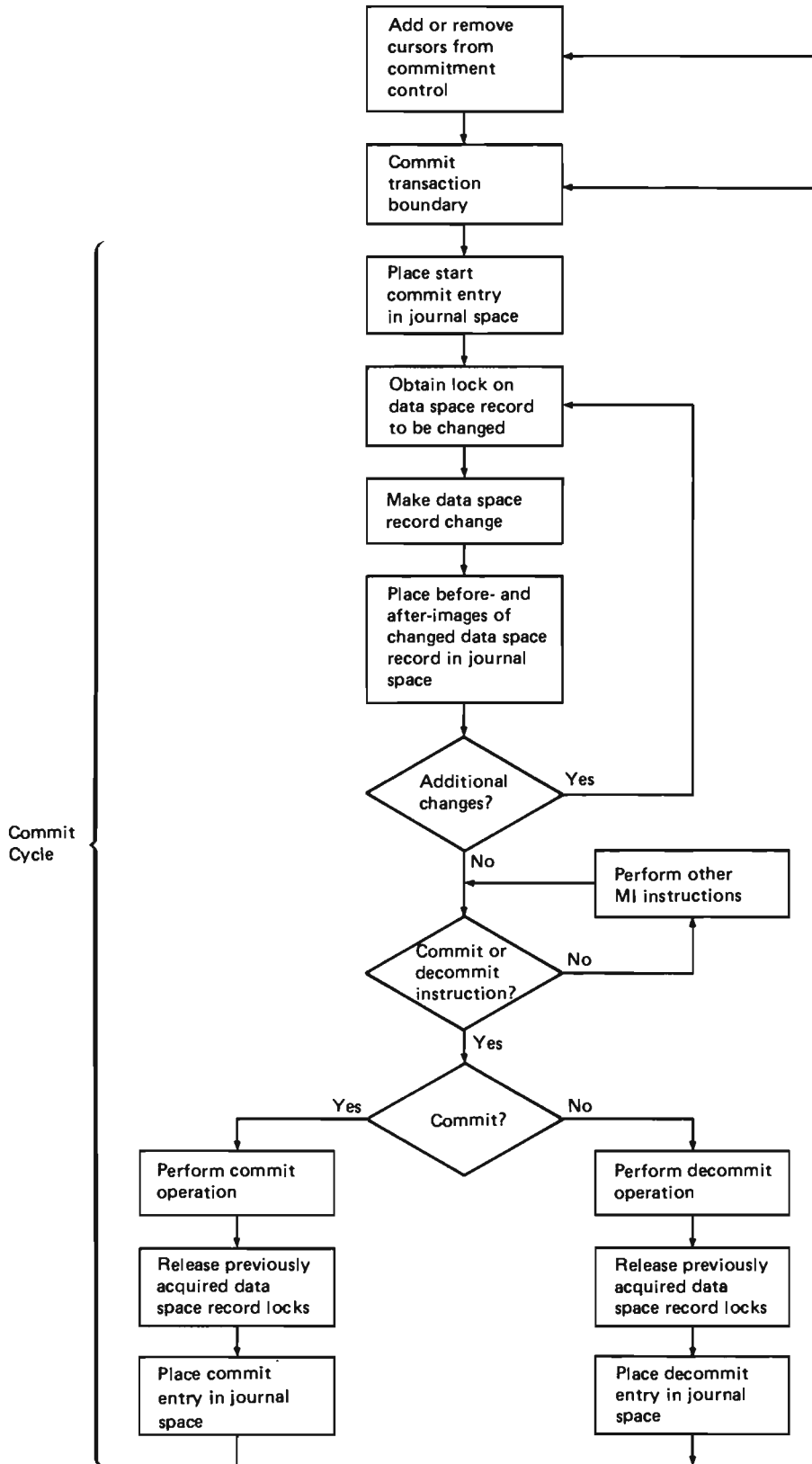
A decommit operation is explicitly performed when the MI Decommit instruction is executed and a start commit entry has been placed in the journal space. If the start commit entry has not been placed in the journal space, no decommit operation is performed.

A decommit operation is implicitly performed during the termination of a process if a start commit entry has been placed in the journal space. A decommit operation is implicitly performed during an IMPL if any process from a prior IMPL has placed a start commit entry in the journal space.

The following occurs when a decommit operation is performed:

- All changes that have not been committed are replaced with before-images. Indexes are maintained, but any error associated with an index results in the index being invalidated and the decommit operation continues. The changes to perform the decommit operation are journaled with an entry subtype, which indicates that the change was caused by a decommit operation.
- A decommit entry is placed in the journal space.
- The data space index keys that were reserved to this commit block are removed.
- The data space entry locks that were held by this commit block are released.
- The data space entry locks that were held by all cursors under commitment control are released.
- The position of cursors under commitment control is reset to the position the cursor had when the commit operation started. Those cursors that were placed under commitment control after the commit operation started are reset to the position they had when placed under commitment control. Those cursors that were removed from commitment control before the decommit operation started are not repositioned.
- Each data space modified by the decommit operation is forced to nonvolatile storage because the changes by the decommit operation are not recorded in a cursor. Consequently, the changes are not forced to nonvolatile storage when the cursor is de-activated.
- The in-use count is decremented for any data space that had its count previously incremented by the data base operations performed in the commit cycle prior to the current decommit operation.
- The lock shared update (LSUP) lock is released for any data space that had been previously locked by the data base operations performed in the commit cycle prior to the current decommit operation.
- The journal use-count is decremented.

The following chart shows a summary of the commit and decommit operations.





Index Management

An independent index is an object that provides search functions and automatically arranges data based on the value of that data. The length of entries in an independent index can be either fixed or variable.

An independent index functions like a table that can be searched for a specific value. It is designed to minimize the:

- Search time to find an entry
- Space that the entry occupies
- Time required to insert an entry
- Time required to delete an entry

USES FOR INDEPENDENT INDEXES

An independent index can be used wherever there is a requirement for table searching, argument–function association, cross referencing, and ordering of data.



Searching for Index Entries

An independent index provides many search functions. More than one index entry can be returned in the user's program as a result of the search operation with the maximum number of entries to be returned specified by the user.

The following types of searches can be performed on an independent index:

Search Type	Result of Search
Find equal	All entries equal to the search argument
Find greater than	All entries greater than the search argument
Find greater than or equal	All entries greater than or equal to the search argument
Find less than	All entries less than the search argument
Find less than or equal	All entries less than or equal to the search argument
Find first	The first (least) entry in the index
Find last	The last (greatest) entry in the index
Find between (inclusive)	All entries between two search arguments

The search argument size can be less than the size of the index entries (referred to as a generic or partial search argument). With generic search arguments, the leading (leftmost) portion of the index entry is matched against the search argument.

For all types of searches, one or more index entries can be returned to the requestor; however, the number of index entries that are returned depends on how many entries match the search criteria. But this number can never exceed the number of entries specified by the requestor. The returned index entries are always the complete entry that was originally inserted into the independent index. A count of zero is returned if a match is not found.

The entries matching the search criteria are either returned to the user (find independent index entry) or removed from the index and optionally returned to the user (remove independent index entry).

INSERTING INDEX ENTRIES

One or more entries can be inserted into an index by the Insert Independent Index Entry instruction. (The maximum length of an index entry is 120 bytes.) An entry cannot be inserted, however, if a duplicate entry of the same length and value already exists in the index.

When an index is created, the leading portion of an index entry can be specified as the key. Searching the index is not limited to the key because any leading portion of the entry can be used against the search argument. But the key is useful for inserting values. Options on insert allow the key portion of the entry to make the insert conditional. When a key is specified, the options are:

- If the key portion of the argument is already in the index, the argument is not inserted into the index.
- If the key portion of the argument is already in the index, the nonkey portion of the argument replaces the nonkey portion of the entry in the index.

An entry can consist of a combination of pointers and data (for example, a descriptor and a pointer to related data in an associated space, or only data). In addition, pointers can appear in the key or nonkey portion of the entry. However, because the value of a pointer is not known, unpredictable results can occur if it is used as a search argument. Pointers cannot be inserted into independent indexes that contain variable-length entries.

PERFORMANCE CONSIDERATIONS

The Modify Independent Index instruction can turn the immediate update attribute for independent indexes on or off. The performance of the system may be increased if this attribute is turned off when many updates are being done on an index.

Queue Management

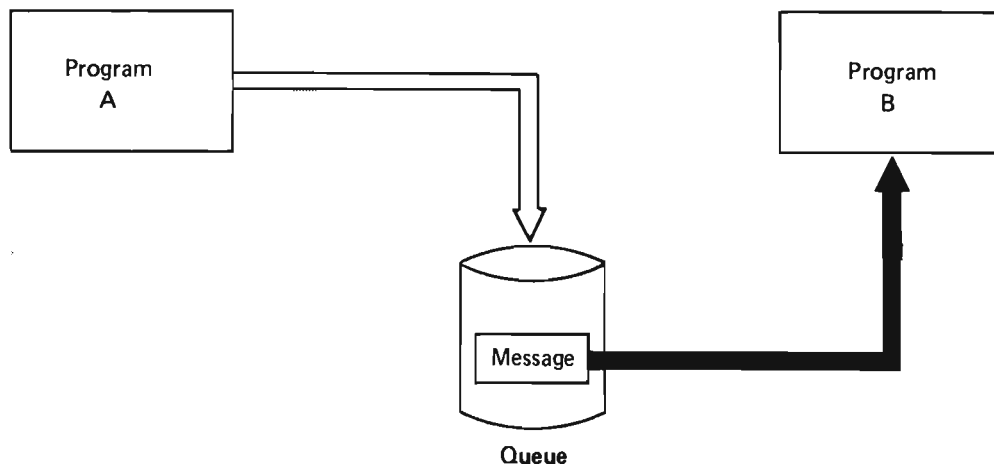
Queues are system objects used to pass information between processes or to synchronize processing between processes.

The information transmitted to a queue is contained in a message. A process builds a message and enqueues it to a queue. Enqueuing a message implies that a copy of the message is stored in the queue in some ordering specification. Any process wishing to access the information in the message, dequeues the message from the queue. Dequeuing the message implies locating the desired message entry, copying the message into a user work area, and removing the message entry from the queue.

In the following illustration, a message is built by process 1 and enqueued to a queue; process 2 dequeues the message.

Process 1

Process 2



When a message is enqueued, there may or may not be a process waiting for that message. If no process is waiting for the message, the message is stored in the queue for later retrieval by some process. When one or more processes are waiting for the message, only one process can receive it; the remaining processes continue to wait for subsequent messages.

When a process attempts to dequeue a message from a queue, a message may or may not be there. If the message is there, it is dequeued; if it is not there, the process may be placed into a dequeue wait state or may immediately continue execution in the program based on optional branch or indicator settings.

QUEUES

A queue is a system object that has message elements enqueued to it and then dequeued from it by processes. Queue messages can be enqueued and dequeued in a sequence defined as an attribute of the queue during queue creation. The sequence can be FIFO (first-in-first-out), LIFO (last-in-first-out), or keyed.

First-in-first-out and last-in-first-out queues order the message entries based on the time of arrival.

When a queue has the keyed attribute, each message contains a key element. The message key is used to order the message entry relative to other messages in the queue. Each message key is treated as though it were character data. When pointers are contained in the message key, they are operationally treated as character scalar data; that is, byte values are used but any pointer characteristics are ignored. It is not possible to take a pointer view of the key that results from a message dequeue. All message keys for a keyed queue must be the same length.

All messages in a queue must be less than or equal to some user-specified size. This value is specified during queue creation.

A queue contains an attribute that specifies the number of messages that can be enqueued at any one time. Another queue attribute defines the action the machine is to take when the current message limit is exceeded; the queue can be implicitly extended to contain more messages, or the queue message limit exceeded exception and event can be signaled when the enqueue is attempted.

Another attribute of a queue identifies whether messages on the queue may contain pointers. If the queue contains messages without pointers, a pointer contained in an enqueued message loses its pointer characteristics when dequeued.

Queue Instructions

The three instructions in queue management used to establish and maintain queues are Create Queue, Materialize Queue Attributes, and Destroy Queue.

Creating a Queue

The Create Queue instruction establishes a system object called a queue (the maximum size of a queue is 16 777 216 bytes). The following information must be provided as attributes of the Create Queue instruction:

- A message content indicator that identifies the presence of pointers in the message text.
- A queue type indicator that identifies the queue as keyed, LIFO, or FIFO. This information establishes the basic sequence in which messages will be received from the queue.
- Maximum number of messages that can be enqueued to the queue at any one time.
- A queue overflow action indicator that specifies the action to be taken if the maximum number of messages is exceeded.
 - If the indicator specifies a fixed size queue, a queue message limit exceeded exception is generated during an enqueue attempt.
 - If the indicator specifies an extendible queue, the maximum message value is extended, and a queue extended event is signaled.
- An extension value that specifies the increment to the maximum number of messages. This value is specified only if the overflow action indicator specifies queue extension.
- A key length that specifies the size of the key for the queue. For a keyed queue, this value must be greater than zero and not more than 256.
- A maximum size of messages to be enqueued that specifies the largest number of bytes that can be contained in any one message enqueued to the queue. For optimum performance, this value should not be larger than necessary for the messages to be enqueued.

Materializing Queue Attributes

The Materialize Queue Attributes instruction allows the user to determine the attributes that were specified when the queue was created. In addition, the current maximum number of messages allowed on the queue and the current number of messages on the queue can also be determined. This information is used to identify the status of a queue at a specific time.

Destroying a Queue

The Destroy Queue instruction removes the queue from the machine. All messages currently enqueued to the queue are also destroyed. All processes currently in a dequeue-wait on the queue are notified, via the object destroyed exception, that the queue has been destroyed.

Any subsequent reference to the queue results in an object destroyed exception.

Queuing Functions

Each message to be enqueued to or dequeued from a queue has a specific format. The format contains two parts, the message prefix and the message text.

Enqueuing Messages

When a message is to be enqueued, the message prefix and the message text must provide the following information:

- **Message prefix**
 - **Size of message text:** The number of bytes to be contained in the queue message entry.
 - **Message key:** The message key used for ordering the message entry relative to other entries in the queue. The length specified in the key length attribute (defined during queue creation) is assumed by the instructions. This entry is required for keyed queues.
- **Message text**
 - The message text is a byte string containing the information to be stored in the queue entry. There is no restriction on the scalar data attributes or pointer types that make up the message text. The length must not exceed the message text length specified in the maximum size of messages to be enqueued attribute specified during queue creation. (The maximum size of a queue message is 64 K bytes.)

Dequeuing Messages

When a message is to be dequeued, the user must provide a message prefix area and a message text area. During the initiation of the Dequeue instruction, the message prefix must be initialized with some information that is required by the instruction. After a successful dequeue, the message prefix and the message text areas contain the following:

- **Message prefix**
 - **Time of enqueue (machine-supplied):** The time stamp value supplied by the machine when the message was enqueued.
 - **Dequeue wait time-out value (user-supplied):** The maximum amount of time the instruction should wait for a message entry to be dequeued before signaling the dequeue-wait-time-out exception.
 - **Size of message dequeued (machine-supplied):** The actual number of bytes contained in the dequeued message.
 - **Access state modification and multiprogramming level options (user-supplied):** Allows the user to control allocation of main storage and processor resources to other processes should the process enter the wait state until a message is available to the process.
 - **Search key (user-supplied):** The key value to be searched for in a keyed queue.
 - **Search criteria:** The specification of how the search key is to relate to the actual key in a queue entry (for example, equal, greater than, or less than). The search key and the entry key are treated as unsigned binary bit strings for the operation.
 - **Actual message key (machine-supplied):** The value of the key associated with the message entry actually dequeued.
- **Message text**
 - The message text is a byte string that contains the message entry after a successful dequeue (scalars and pointer data). If pointers are contained in the message text, the queue must have been created with the attribute that permits pointers in the message text and the message text area must be 16-byte aligned in the space. Otherwise, a pointer contained in an enqueued message is dequeued as scalar data.

Moving Messages

Two instructions (Enqueue and Dequeue) are used to move messages between queues and processes. The Enqueue instruction causes a message to be placed on a queue and allows any process that may be waiting for the message to be made eligible for dispatching. The Dequeue instruction removes a message from a queue or causes a dequeue wait until a message is available.

Enqueuing a Message

The Enqueue instruction causes a message to be enqueued to a queue and makes any process waiting for the message eligible for processor resources (subject to multiprogramming level constraints). Note that if a message satisfies more than one process waiting for a queue, only the highest priority process waiting the longest receives the message. Any other process continues to wait for the next message.

The ordering of messages in the queue is determined by the queue attributes (LIFO, FIFO, or keyed) and the order of enqueue or the key values used for each Enqueue instruction. If keyed sequence is specified, the messages are ordered in ascending sequence, and last within equal key. Ordering of keys is in binary collating sequence. FIFO and LIFO queues can contain keys, but the keys are considered only additional text and do not enter into the enqueue/dequeue criteria. Keyed queues may only be operated on with key values.

Note that for a given queue, all message keys are considered to be character data and of identical length.

The message text is specified as an operand on the Enqueue instruction. The Enqueue instruction copies the data fields specified in the message text source operand.

Dequeuing a Message

The Dequeue instruction allows a process to remove a message from a queue.

If a message cannot be found that satisfies the dequeue criteria, the process is put into a dequeue wait until either a message arrives to satisfy the Dequeue instruction or the dequeue wait time-out value is exceeded. If a dequeue wait time-out occurs, a dequeue wait time-out exception is signaled to the waiting process. The Dequeue instruction allows an override to the dequeue wait time-out value specified as the process default wait time-out. If no wait time-out value is specified (in either the Dequeue instruction or as a process attribute) an immediate wait time-out exception is signaled.

If the Dequeue instruction is used with branch or indicator options and no message is on the queue to satisfy the dequeue criteria, the process is not put into a dequeue wait. Instead, control is given to the instruction specified as a branch target in the Dequeue instruction, or the specified indicators are set. This procedure allows polling for messages on multiple queues without having to wait.

The Create Queue instruction can create a queue that allows retrieval by keyed selection. The keyed option allows retrieval by any of the following relations between a message key and a user-specified compare operand:

	Relation	
	>	
	≥	
	=	
Message Key	<	Compare Operand
	≤	
	≠	

Materialize Queue Messages

The Materialize Queue Messages instruction can materialize one or more messages on a queue.

The amount of key and message text data materialized for each message is controlled by a message selection template.

Space Management

A space is a contiguous set of bytes that provides a storage area for data objects (scalar and pointer). A space can be contained in any system object and its contents addressed and manipulated on a byte-by-byte basis. The creation template of the object allows the user to specify the size of the space to be allocated for the object.

All system objects except the space object serve other functions in addition to containing a space. The space object consists of only the associated space and provides no other function.

A space contains a contiguous string of 8-bit bytes. The maximum length of a space is dependent on the system object it is associated with, up to a maximum of 16 777 184 bytes. Data views can be defined in order to superimpose certain attributes onto the data at desired locations within a space. These data views have representational as well as certain operational characteristics that allow them to be referred to as operands in System/38 instructions. The data for an operand resides in the space, but the attributes for the data are determined by the data view definition.

SPACES

Spaces exist in association with system objects. All system objects may optionally have an associated space. Each system object then, has a functional capability plus an additional capability of containing a space. A space object is unique in that it has no other function other than the capability to contain a space. By associating a space with a system object, the user has the capability to store data values associated with the object in an area addressable through the system object itself (a system pointer or a symbolic address); that is, once the object is located, its data values are available.

All spaces (those in space objects as well as those in other system objects) have the following characteristics:

- Spaces can be fixed or variable in size. Variable-length spaces can be extended or truncated by the Modify Space instruction. Fixed-length spaces remain the size that was specified during their initial allocation.
- The size of the allocated space is at least as large as requested. This size is dependent on the packaging of the object that contains the space. When the object size is materialized, the space size reflects the current allocation, not the requested size. All of the allocated bytes can be used for data object manipulation.
- When spaces are allocated or extended, each byte of the allocation is given an initial value that was specified at object creation time. (Spaces are extended in multiples of 512 bytes.) If the object has no space, the initial value attribute is ignored.
- The maximum size of a space is dependent on the type and characteristics of the object containing the space; the maximum size is 16 777 184 bytes.

SPACE FUNCTIONS

As previously indicated, spaces can be created as independent system objects. As such, there are instructions that operate only on these objects. For example, the Create Space, Destroy Space, Materialize Space Attributes, and Modify Space Attributes instructions refer to spaces rather than the byte strings contained in the spaces.

These instructions address space objects through a system pointer. Once a system pointer addresses a system object (in this case, a space object), the system pointer assumes the attributes as well as the address of the system object. A future reference to the system pointer ensures that the system object referred to is of the proper type.

Space Creation

The Create Space instruction creates and allocates a space system object according to the attributes specified in a template operand.

The space contains the number of bytes specified in the creation template. Unlike other system objects that contain spaces, a space object has no other function.

Addressability to the newly created space is returned in the system pointer specified in the Create Space instruction. Future references to the space object are made through the system pointer. Individual bytes of the space can be referenced through a space pointer.

Space Attribute Materialization

The Materialize Space Attributes instruction can be used to materialize the attributes of a space object into a string in order to determine current attributes.

The attributes of a space associated with a system object can be materialized either through use of the Materialize instruction for the system object it is associated with or through the use of the Materialize System Object instruction.

Space Attribute Modification

The Modify Space Attributes instruction changes the various attributes of the space associated with a system object. Attributes that can be changed include the initial value of the space, whether or not the space is allocated any space storage, and whether or not the size of the space can be changed.

If the space is extended, the bytes are initialized to the initial value specified for the space.

Space Destruction

A space system object can be destroyed by the Destroy Space instruction. Future attempts to refer to the space through the system pointer result in an object destroyed exception. Addressability to the space is removed from the addressing context (if there is one).

SPACE DATA

Space Data Views

Space data views provide a means of defining and using scalar and pointer data objects within a space. Data views are defined in the ODT (object definition table) of a program.

Data views provide a logical mapping of certain attributes onto a portion of a space. The attributes provide the data view with representational and operational characteristics. In addition to the attributes, the data view logically consists of addressability to a byte string in the space. A reference to a data view logically associates the attributes with the byte string.

Data views imply the following for spaces:

- For scalar operands, the attributes define the representation (format) for the area defined by the data view. For source operands, the byte values are assumed to have the required format (for binary and character data) or are verified to ensure that valid data exists at that location (for zoned and packed decimal data). For receiver operands, the attributes define the format of the data to be stored at that location.
- For pointer operands, the attributes specify that the 16 bytes defined by the area are to be considered to contain a pointer. For source operands, the area is verified to ensure that a pointer of the proper type is contained in the area. For receiver operands, a pointer of the proper type is constructed in the area. Storing scalar data into an area containing a valid pointer causes the machine to no longer consider the area to contain a pointer.

Space Addressing

Addressability of data views consists of two parts: which space is being referred to and the location (offset) into the space that is to be referred to. The first byte in the space has an offset value of zero.

Each byte in the space is separately addressable and can be referenced through a data view defined in the program. The reference can be to a location relative to: the allocation of static or automatic storage for a program within a process, the space addressability contained in a space pointer, or to an argument data object passed from a preceding program.

Dump Space Management

Dump Space Management provides the MI user with an online storage medium (a dump space) for a dump of system objects. It also provides the MI user with the ability to perform simple manipulation of data contained in a dump.

A dump of system objects can be put into a dump space through a source/sink dump operation. System objects within a dump in a dump space can be loaded back on the machine through a source/sink load operation.

The dump data in a dump space can be retrieved and inserted into another dump space. The target dump space can exist on the system where the dump originated, or on a system where the dump data was transmitted.

Dump Spaces

A dump space is a system object that serves as a storage area for a dump of other system objects. It provides an online storage alternative to the commonly used offline storage media for dumps of system objects.

A dump space contains a storage area for a contiguous string of 8-bit bytes. The storage area varies in size, but is no larger than 2 013 204 480 bytes, which equals almost 2 gigabytes. The user can first specify the size of a dump space when the space is created. The space size can be internally extended by the machine for dump and insert operations and can be externally reset through a modify operation.

Dump space objects store only dump data; unlike data spaces and space objects, which store any type of general data.

Dump Space Functions

The following instructions help manipulate the dump space, not the dump data which may be contained within the space.

Dump Space Creation

The Create Dump Space instruction creates and allocates a dump space system object according to the attributes specified in a template operand. The dump space can then be used as a storage area for a source/sink dump of system objects.

Addressability to the newly created dump space is returned in the system pointer specified on the instruction. Future references to the dump space are made through the system pointer. The dump data put into a dump space through a source/sink dump can be manipulated through insert and retrieve operations defined later in this section.

Dump Space Materialization

The Materialize Dump Space instruction can be used to materialize the attributes that relate specifically to a dump space into a string in order to determine their current value.

The Materialize System Object instruction can be used to materialize common system object attributes of a dump space in order to determine their current value.

Dump Space Modification

The Modify Dump Space instruction can be used to modify certain attributes that relate specifically to a dump space. The allocation size of the dump space can be reset back to the size of the dump data contained in it. The dump space can be reset to a state indicating that it contains no dump data.

Dump Space Destruction

The Destroy Dump Space instruction destroys a dump space and frees up the storage allocated to the object. Future attempts to refer to the dump space through the system pointer result in the object destroyed exception. Addressability to the dump space is removed from the addressing context (if there is one).

Dump Space Data

The format and meaning of the dump data contained within a dump space is not defined, other than to provide for its retrieval from a dump space and subsequent insertion into a dump space. Dump data is initially set into a dump space through a source/sink dump operation to dump system objects into the dump space. Subsequently, the system objects contained in the dump data can be loaded back into existence in the machine through a source/sink load operation. The format of the dump data produced by a dump operation is an internal characteristic of the specific implementation of the machine and is not defined for the MI user.

Retrieve and insert operations are supported for dump data to provide for movement of the dump data from one dump space to another, where the target dump space can be on a different machine than the source dump space.

Load/Dump Functions

The Request Path Operation instruction can be used to perform source/sink dump or load operations to or from a dump space. A dump operation sets the appropriate dump data into a dump space to back up the current state of the specified system objects in a form that provides for the subsequent loading of them back into the machine. A load operation operates on the dump data produced from a prior dump operation to load the system objects contained in the dump space back into existence on the machine.

The Request I/O instruction can be used to perform source/sink load or dump operations on a dump space. A dump operation saves the dump space to an LD (load/dump) storage media. A load operation restores the dump space from an LD storage media. Objects contained in a dump space dumped to an LD media can be directly loaded from the LD media. See the *Load/Dump Considerations* section, under the *Set Load/Dump Parameter* command in Chapter 6, for more information.

Dump Space Data Retrieval

The Retrieve Dump Data instruction can be used to retrieve dump data contained in a dump space. The retrieval is performed through a simple relative block access of the dump data. The format of the dump data retrieved is undefined, other than its size and that it is packaged with a small amount of additional data used for verifications when the data is inserted into a target dump space.

Dump Space Data Insertion

The Insert Dump Data instruction can be used to place dump data previously retrieved from a dump space into a target dump space. The insertion of dump data is performed in a simple progression of fixed-length blocks of dump data, starting with the first block of data retrieved from the source dump space and continuing in ascending order out to the end of the dump data retrieved.

The format of the dump data to be inserted is undefined other than its size and that it is packaged with a small amount of additional data used for verifications during its insertion. The verifications performed on the data are done to ensure the dump data is valid for the current attributes and usage of the target dump space. These verifications help to ensure machine integrity when the objects are loaded back into the machine and referenced during machine operations on them.

Space Data Modification

The value of a space is the value of the byte strings contained in the space. All or part of the value can be changed with System/38 instructions.

As discussed under *Computation and Branching* in Chapter 3, the computational instructions operate on scalar data objects. These scalar data objects, then, provide for modification of only certain parts of the space value. Even though these data objects have certain representations, when seen as part of a space they are only strings of 8-bit bytes.

As discussed under *Addressing* in Chapter 2, pointers are contained in spaces. The instructions used to manipulate pointers (for example, Set Space Pointer or Resolve System Pointer) cause pointers to be created and stored in spaces. These pointers can then be used in subsequent operations.

Certain instructions, for example, return templates that consist of scalar and pointer data. These templates are located by a space pointer. The first 4 bytes of the template identify the number of bytes that may be modified. These bytes are assigned values based on the characteristics of the materialization.



Source/sink management provides a set of system objects and a set of instructions that operate on those objects. Source/sink objects define the various external I/O devices and the methods of attaching the devices to the system.

The source/sink instructions manipulate and control the use of the I/O devices, manage the attachment network¹ mechanisms, and define the configuration details of the system.

SOURCE/SINK OBJECTS

Source/sink management is based on three fundamental system objects (logical unit description, controller description, and network description) that characterize three basic aspects of how all I/O devices relate to a system.

Figure 6-1 shows the components that make up some of the I/O and communications networks that are available on System/38. These components are used as follows:

- The I/O device or end-use-mechanism is the ultimate object of I/O transactions and is called the LU (logical unit).

¹The term network has several meanings. A *public network* is a network established and operated by common carriers or telecommunications administrations for the specific purpose of providing circuit-switched, nonswitched-circuit, or packet switched services to the public. A *user application network* is a configuration of data processing products (such as processing units or work stations) established and operated by users for the purpose of data processing or information exchange; such a network may use transport services offered by common carriers or telecommunications administrations.

Network, as used in this publication, refers to a user application network except in those cases where it specifically mentions telecommunications networks (for example, X.25 packet switching data networks).

Packet switching data network (PSDN) refers to a communications network that uses packet switching as a means of transmitting data.

- The device controller is either the I/O controller for clusters of I/O devices grouped together or the station that attaches groups of communications devices over the same data link.
- When System/38 is the primary station, the controller:
 - Can be a separate unit, or it can be physically packaged within one of the devices it supports.
 - Has separate characteristics and requirements from the device.
 - Can have a telephone number if it is a station on a switched communications line or can have a remote network address if it is a controller on a packet switching network.
 - Can have a channel address if it represents a channel-attached control unit.
- When System/38 is the secondary station, the controller:
 - Is the host system line controller.
 - Has a control program that controls the communications network to which System/38 is attached.
 - Has separate characteristics and requirements from the host system.
 - Can have a telephone number if System/38 is connected to the host system via a switched communications line or can have a remote network address if it is a controller on a packet switching network.
- When System/38 is a peer station, the controller:
 - Can be a separate unit, or it can be physically packaged within one of the devices it supports, or it can be the system line controller.
 - Has separate characteristics and requirements from the device or system.
 - Can have a telephone number if System/38 connects to the controller via a switched telephone line or can have a remote network address if it is a controller on a packet switching network.

- The network port on a system is the hardware that supports attachment of I/O devices and controllers. The characteristics of a network port must match the characteristics of the controllers or stations that attach to that port. Some of these port characteristics are:
 - Type of modem or hardware interface used
 - Data rates supported
 - Communications protocols used (SDLC, X.25, or BSC)
 - Operating modes
 - Communication role (primary station or secondary station)

The characteristics of the network port for X.25 must match those of the local DCE—not those for the controller. The role of data link for X.25/HDLC is peer; that is, it allows connection of both primary and secondary stations.

For more information about System/38 communications facilities, refer to the *Data Communications Programmer's Guide*.

Object Types

The source/sink objects that correspond to the components of the I/O network (shown in Figure 6-1) are:

- Logical unit description (LUD)
- Controller description (CD)
- Network description (ND)

LUDs are the control objects upon which source/sink management is built. An LUD is directly involved in actual input/output operations and must be created for every I/O device or end-use-mechanism on the system. These objects contain enough information about each device to uniquely identify that device. Typical contents are LUD name, device type, physical address of the device, device features, characteristics, operating parameters, and device status.

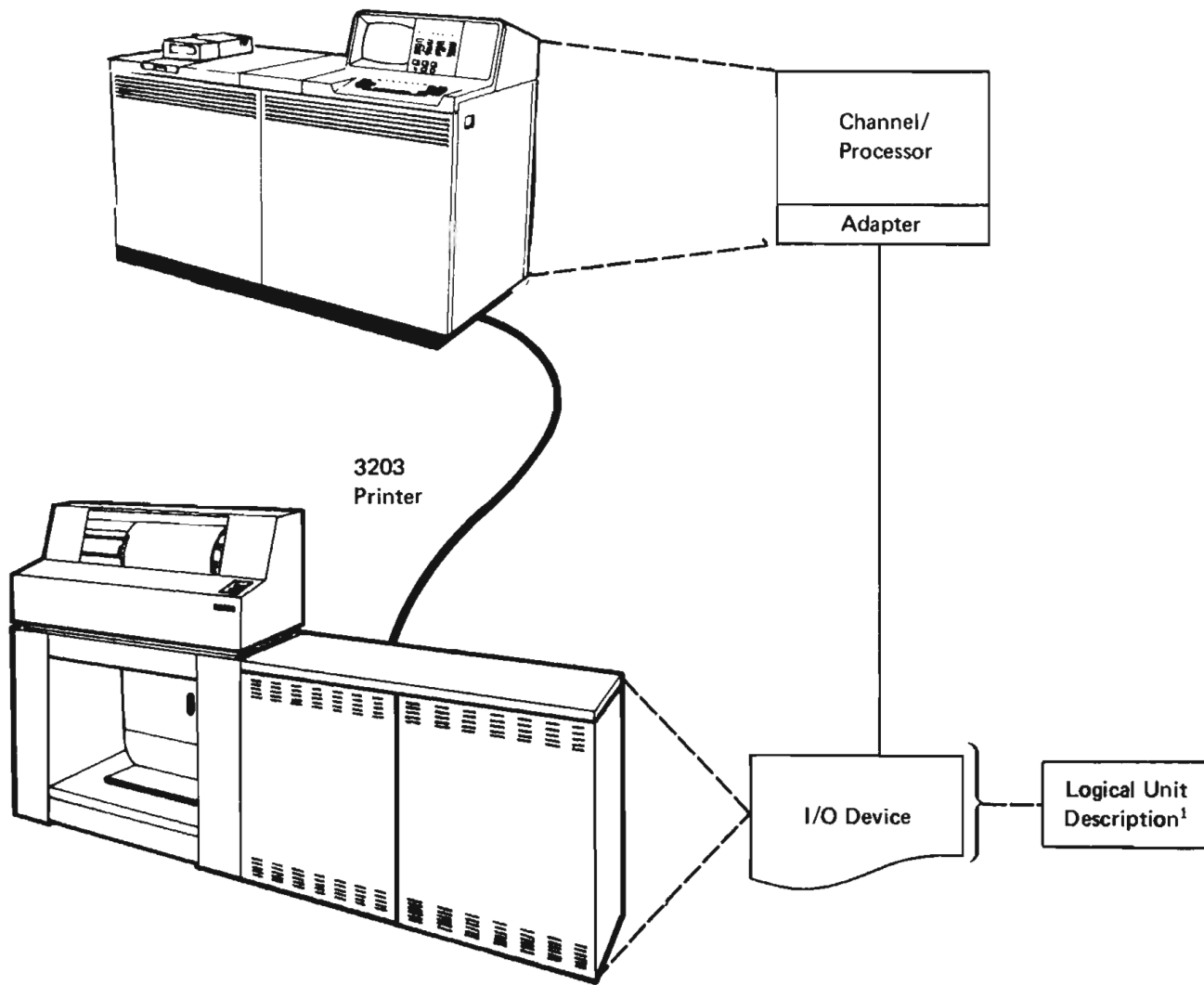
In some cases, the end-use-mechanism is not considered to be an I/O device. Rather, an end-use-mechanism is considered to be the ultimate destination point for the data being transmitted through the I/O network. For example, in the case where a System/38 is defined as being the secondary device and is communicating with a host primary system (such as a System/370), then an LUD represents the end-use-mechanism in the host system. The end-use-mechanism in this case could be an application program in the host system that is communicating with an application program in System/38.

CDs must be created for every communications station or device controller that can be attached to the system (this includes switched communications lines). CDs contain the information to uniquely identify each station or controller. Typical contents are CD name, controller unit type, station address, station identification, operating parameters, and controller status. This information can also be used for authorization and identification procedures. In addition, for X.25, the CD describes the run time features of the virtual circuit used.

NDs must be created for every communications port on the system. Typical contents are ND name, physical address of the port, network characteristics, and status of the port.

Local Device

System/38

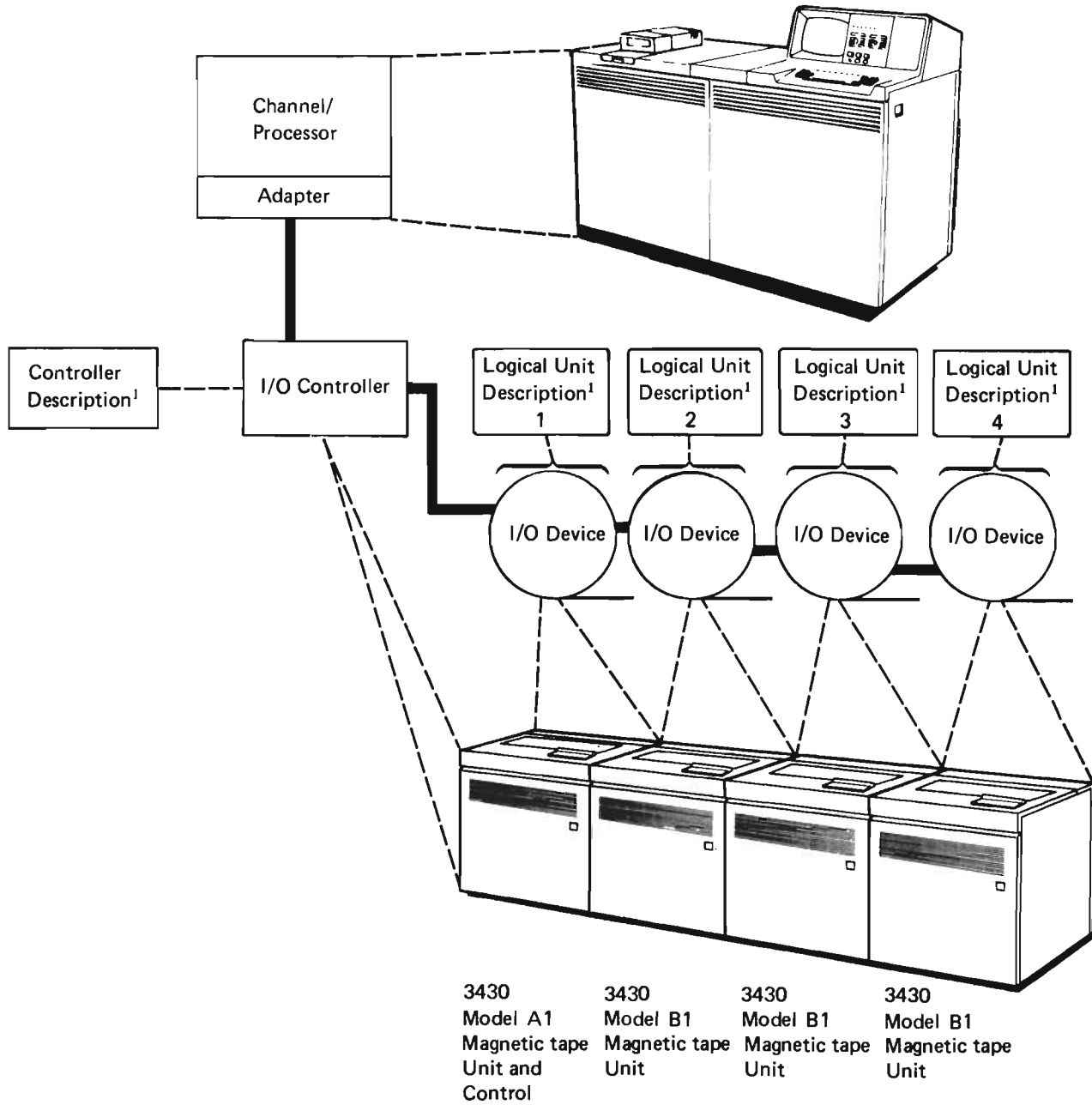


¹This system object in the System/38 programming support represents the corresponding physical element in the I/O network.

Figure 6-1 (Part 1 of 11). Input/Output Network Components

Local Subsystem Devices

System/38

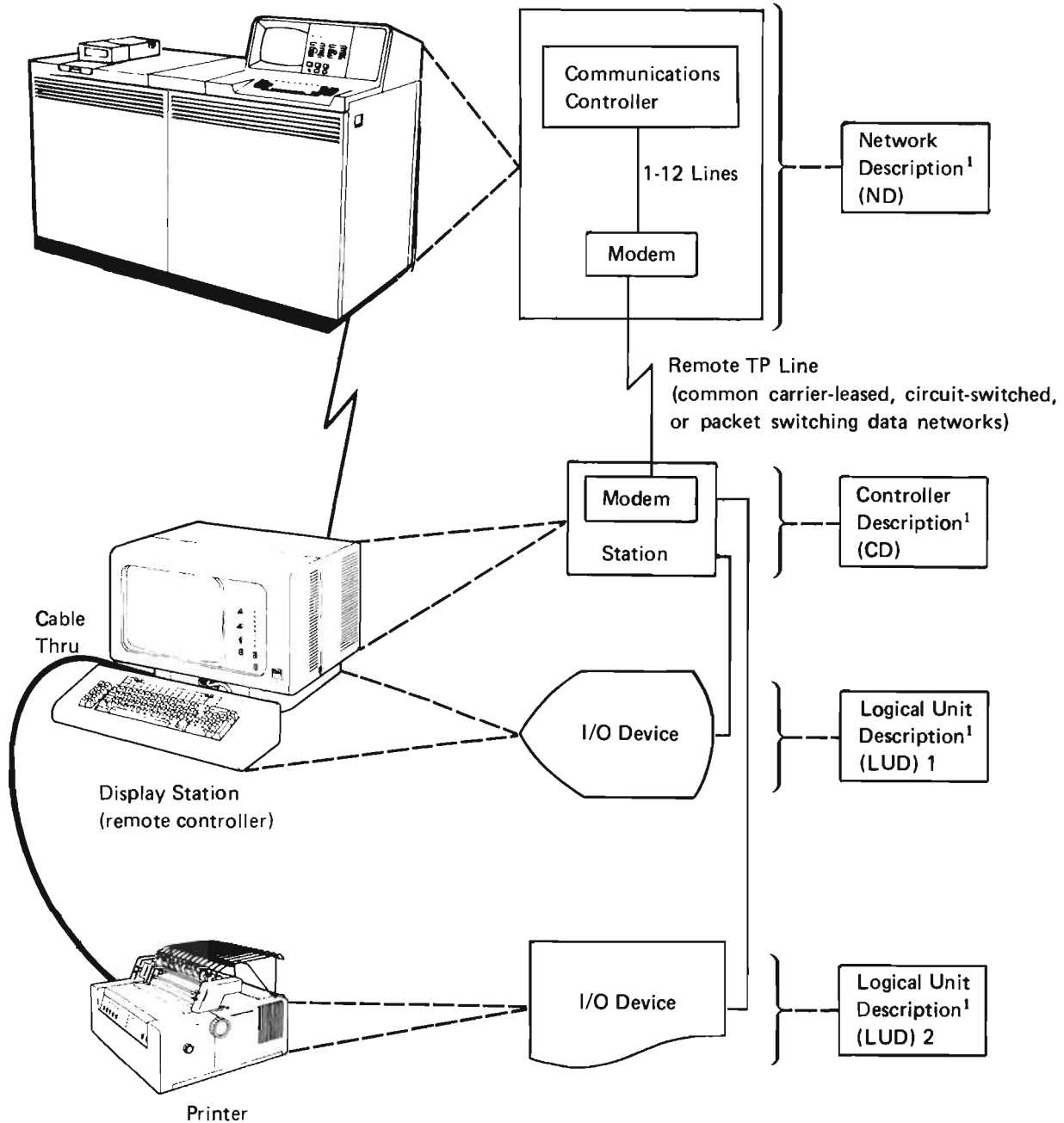


¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 2 of 11). Input/Output Network Components

Remotely Attached Devices

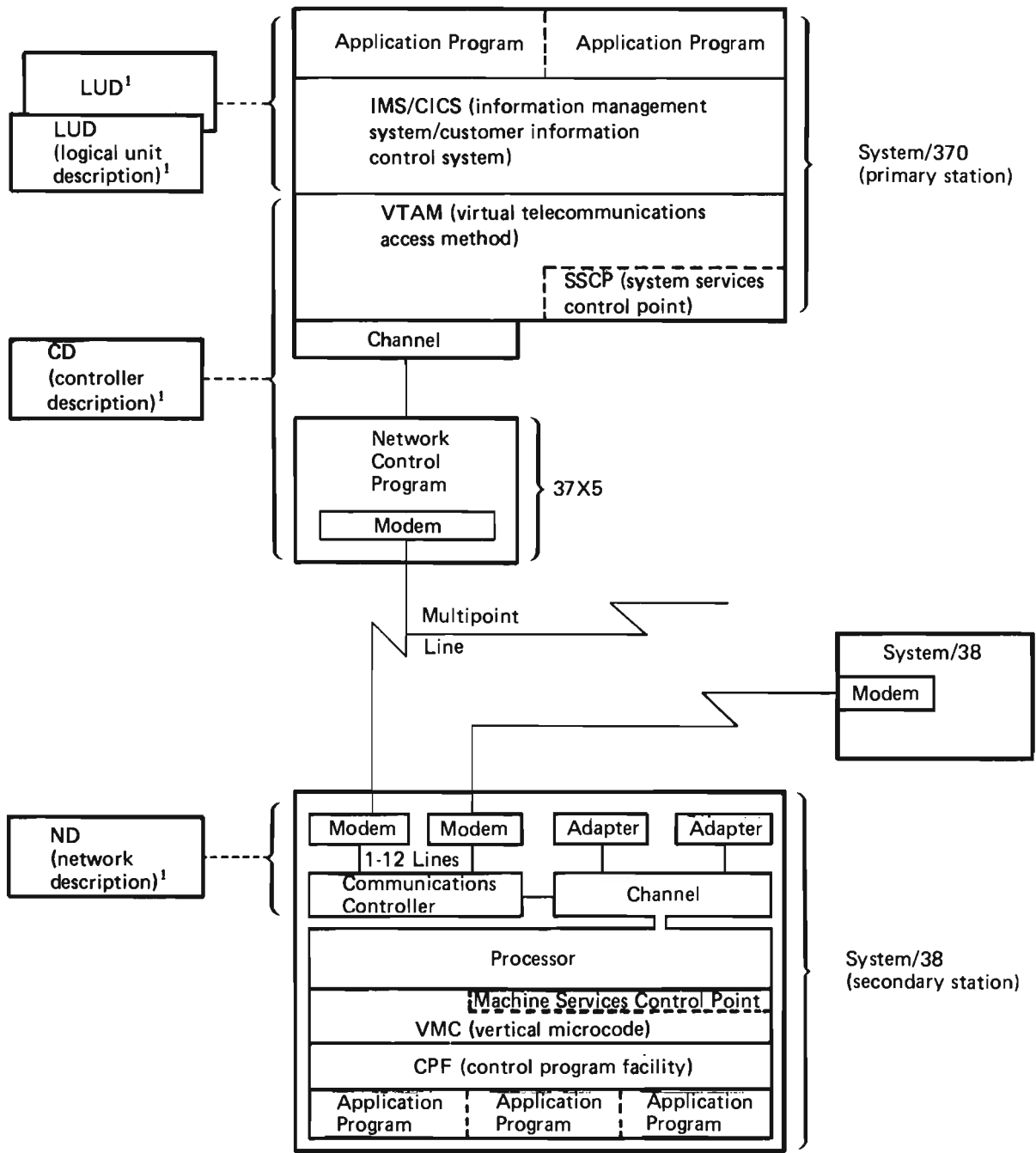
System/38



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 3 of 11). Input/Output Network Components

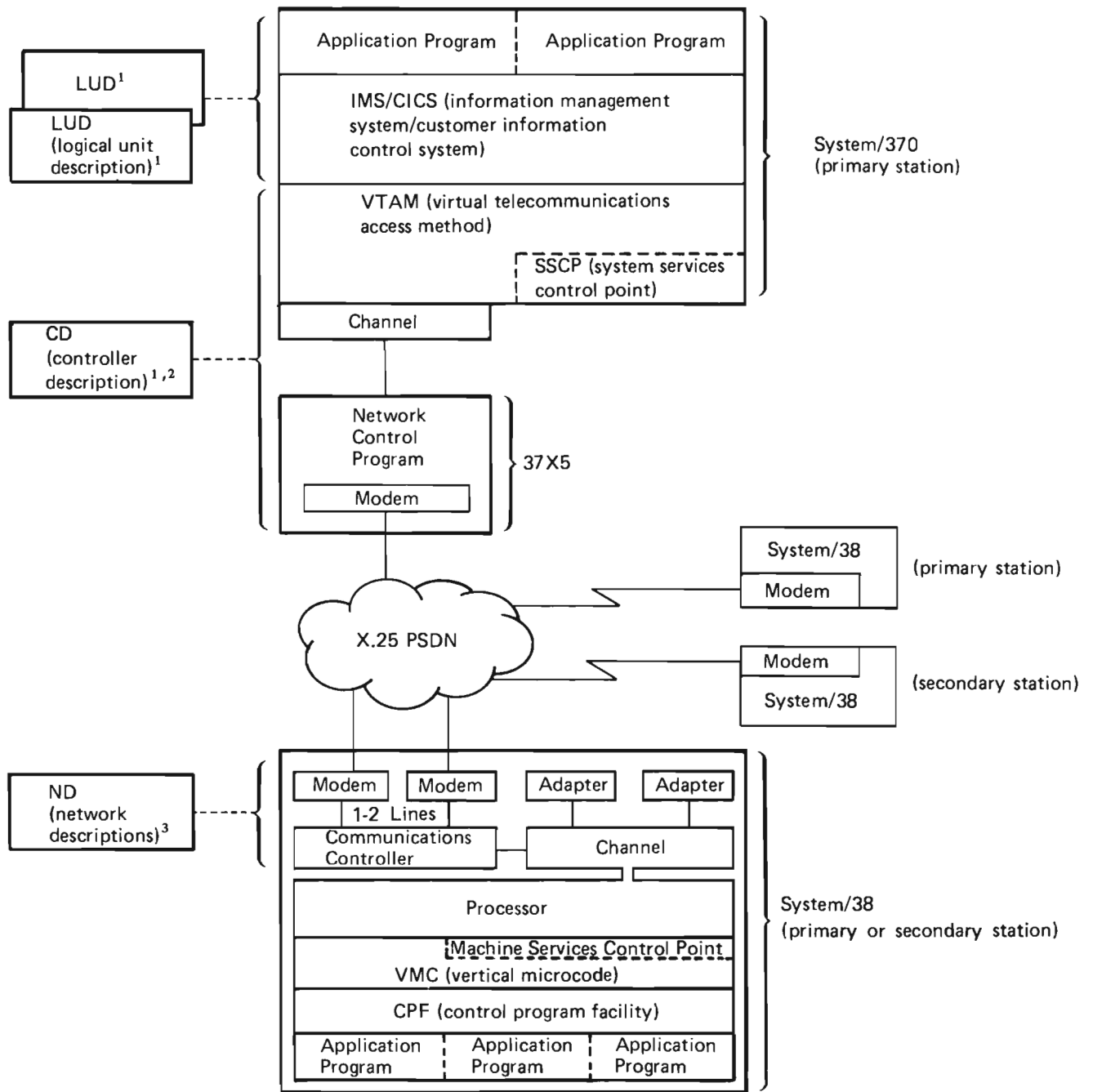
System to System Attachment (SNA)



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 4 of 11). Input/Output Network Components

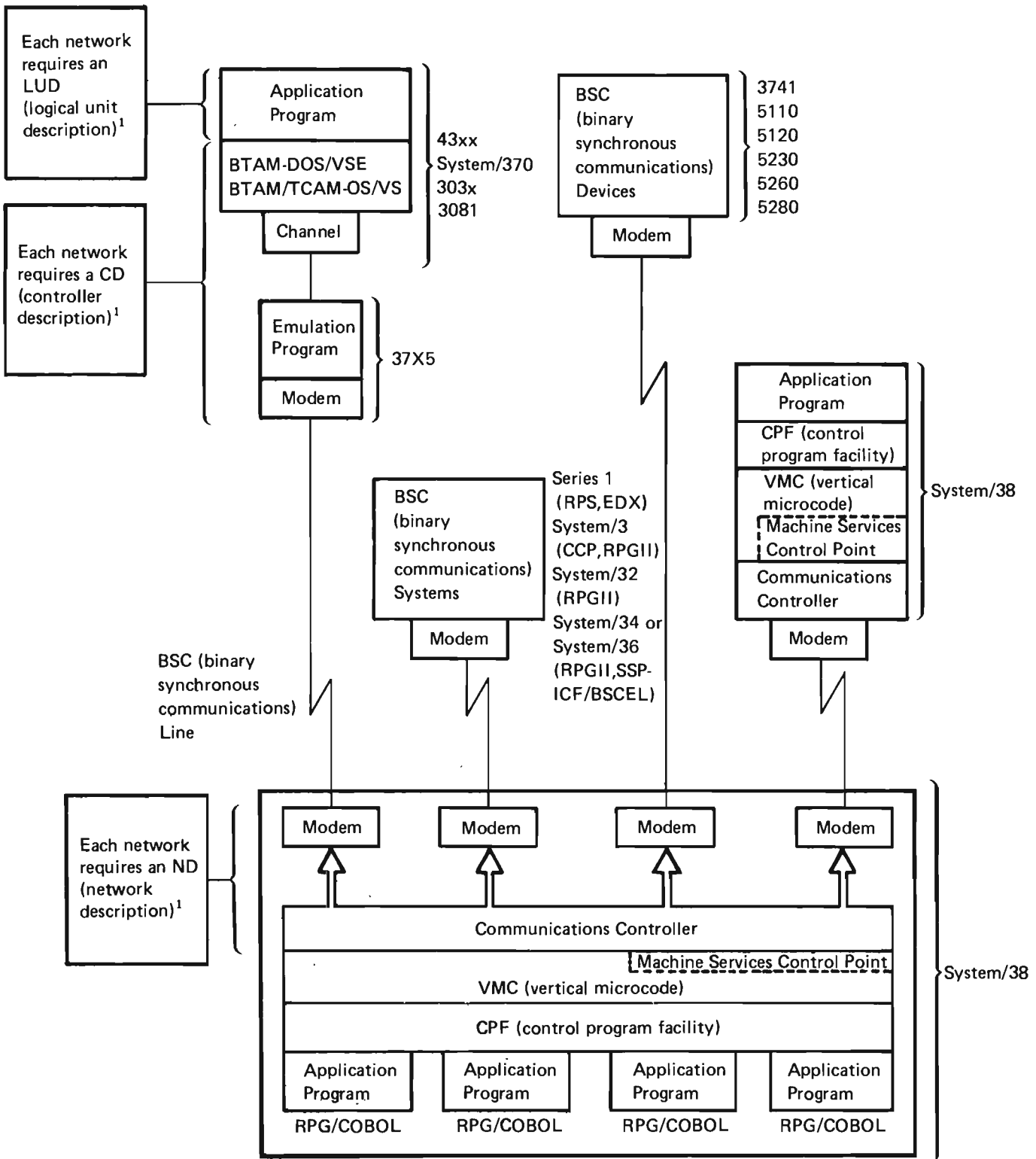
System to System Attachment (Using SNA On an X.25 PSDN)



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.
²The CD for X.25 takes on an additional role. Not only does it describe the remote controller characteristics, it also describes the run time characteristics of the virtual circuit connection to the DCE at the controller's location. This includes user facilities, flow control options, charging options, closed user groups, and so on.
³The ND for X.25 takes on an additional role. Not only does it describe the physical line features to the local DCE, as other NDs do, it also describes the X.25 packet switching data network (PSDN) subscription as agreed to by the PSDN vendor. This includes logical channel descriptions, link start-up procedures, and so on.

Figure 6-1 (Part 5 of 11). Input/Output Network Components

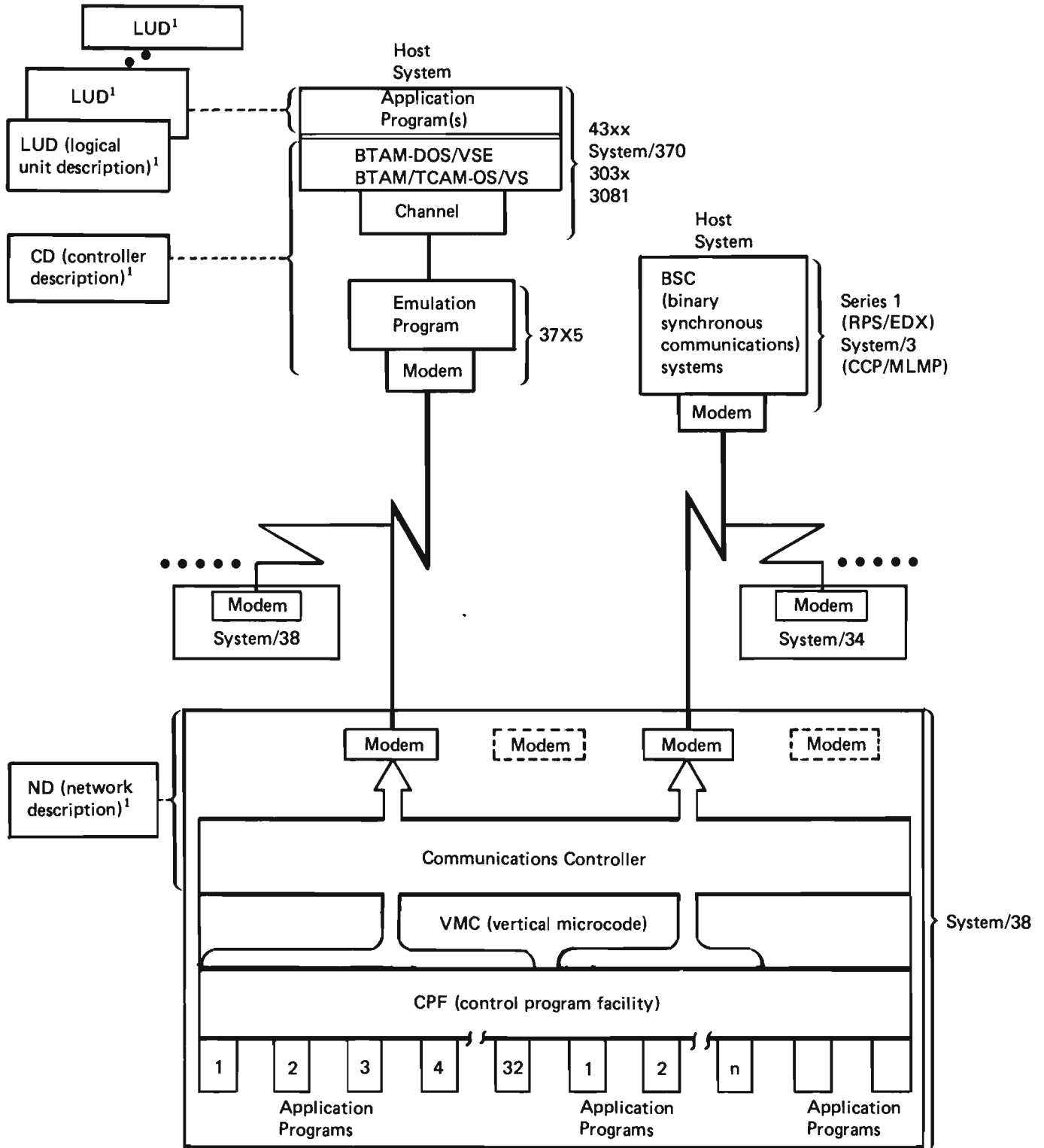
Binary Synchronous Communications Attachments (Point-to-Point)



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 6 of 11). Input/Output Network Components

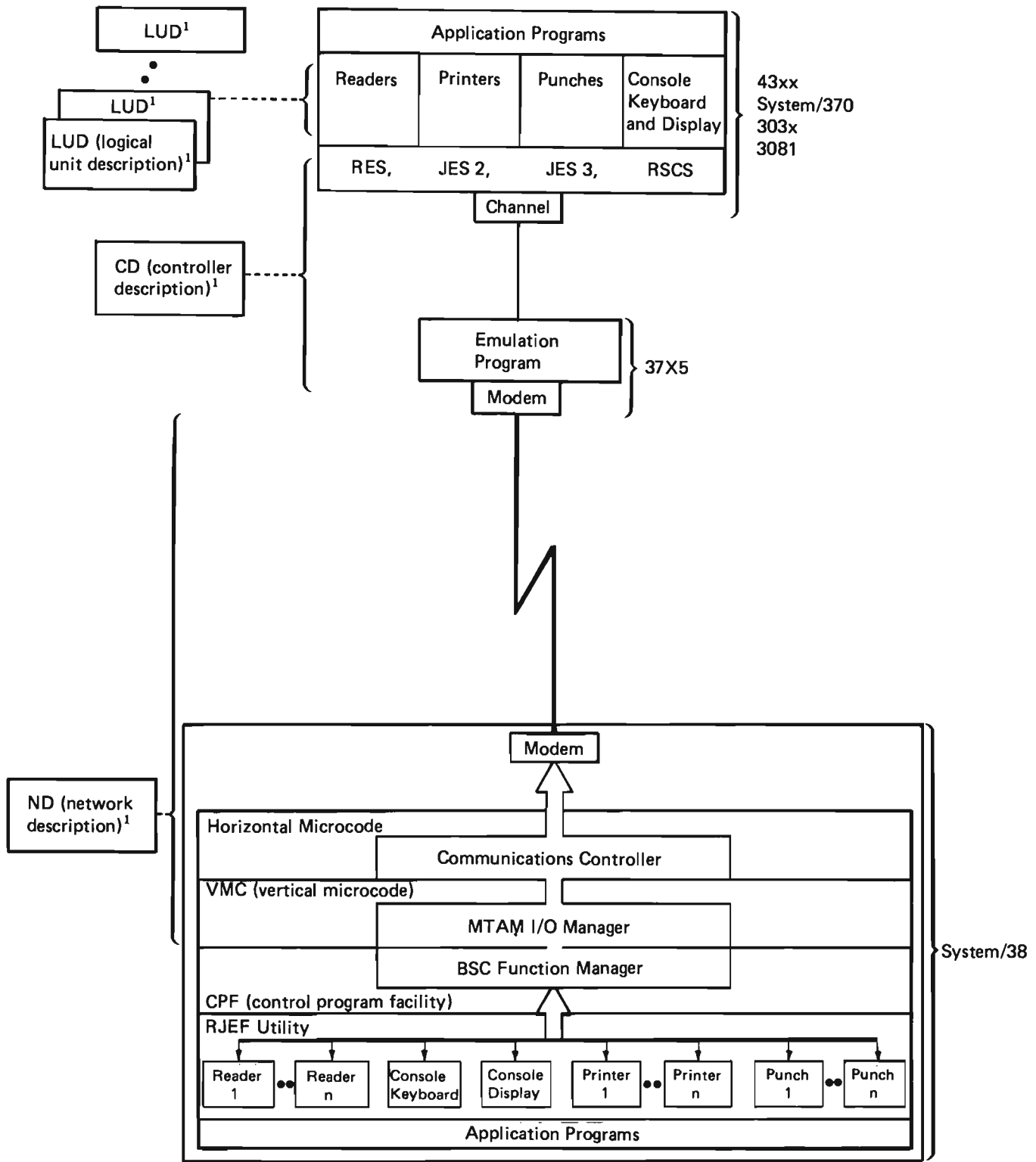
Binary Synchronous Communications Attachments (Multipoint Tributary)



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 7 of 11). Input/Output Network Components

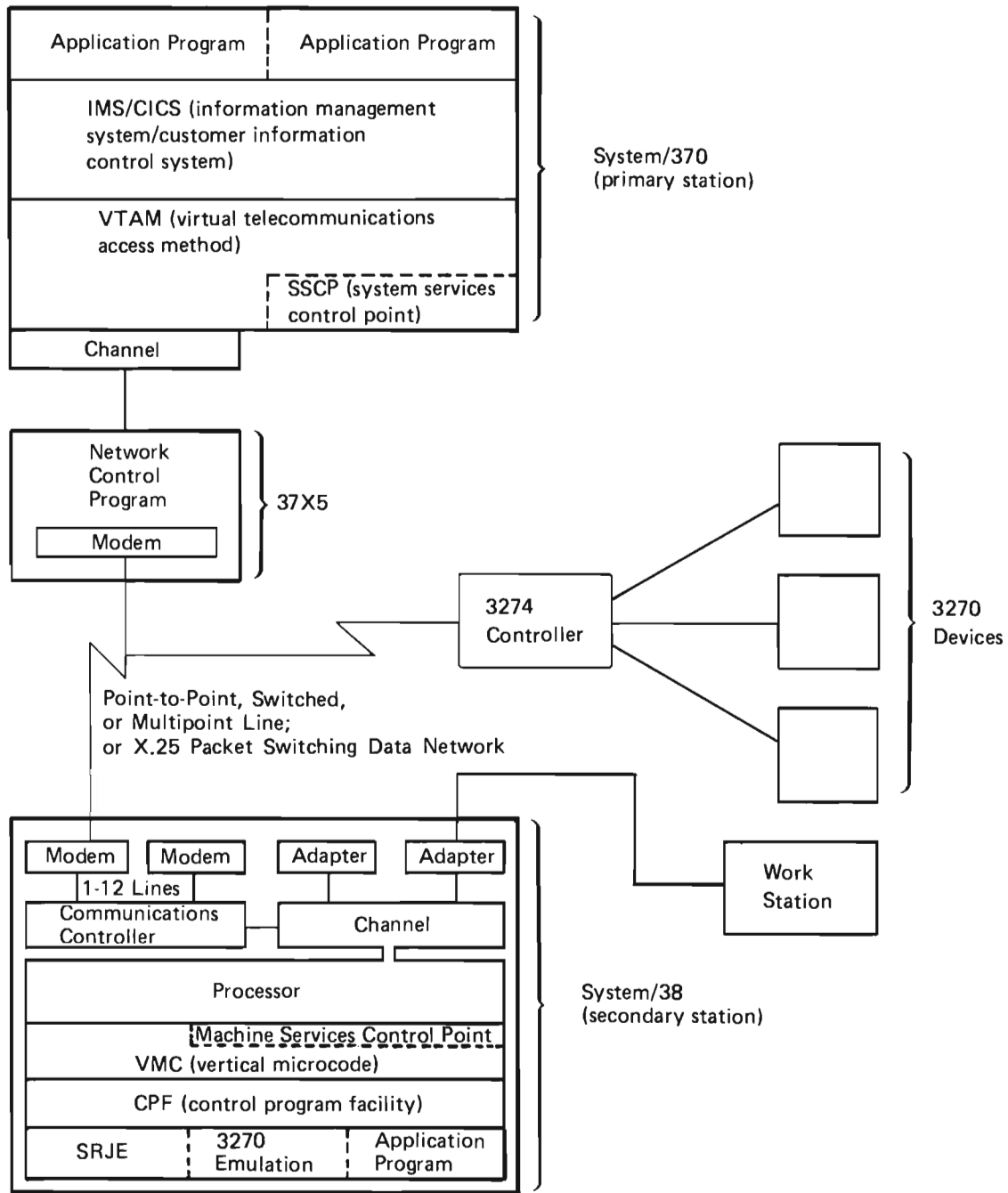
Multi-leaving Telecommunications Access Method Support for MRJE



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 8 of 11). Input/Output Network Components

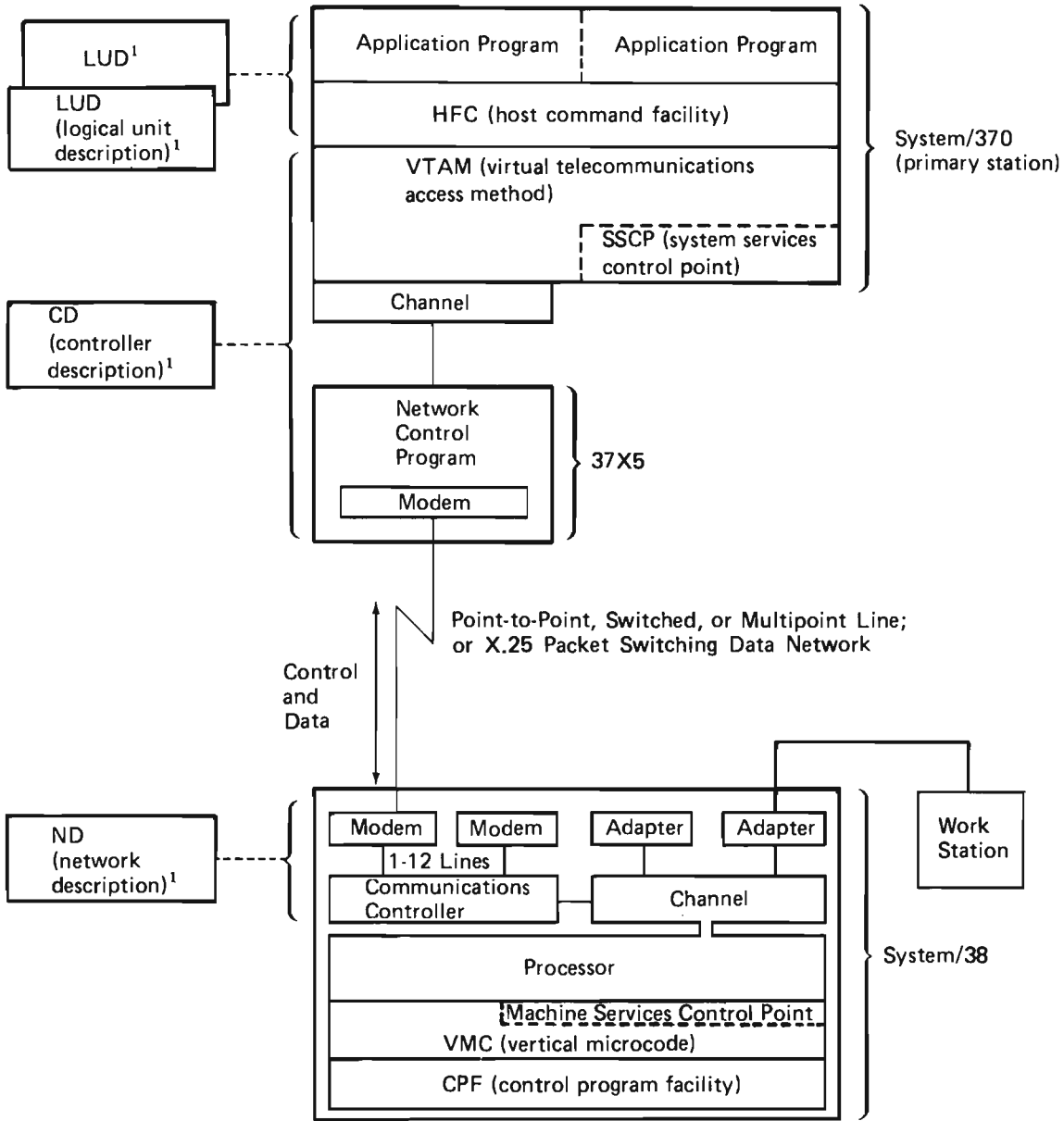
System to System Attachment (SNA)



System/38 can coexist with 3270 devices on the same SNA link and also support multiple sessions of LU-1 3770 emulation; LU-1, LU-2, or LU-3 3270 emulation; SRJE; APPC; and DHCF on the same link.

Figure 6-1 (Part 9 of 11). Input/Output Network Components

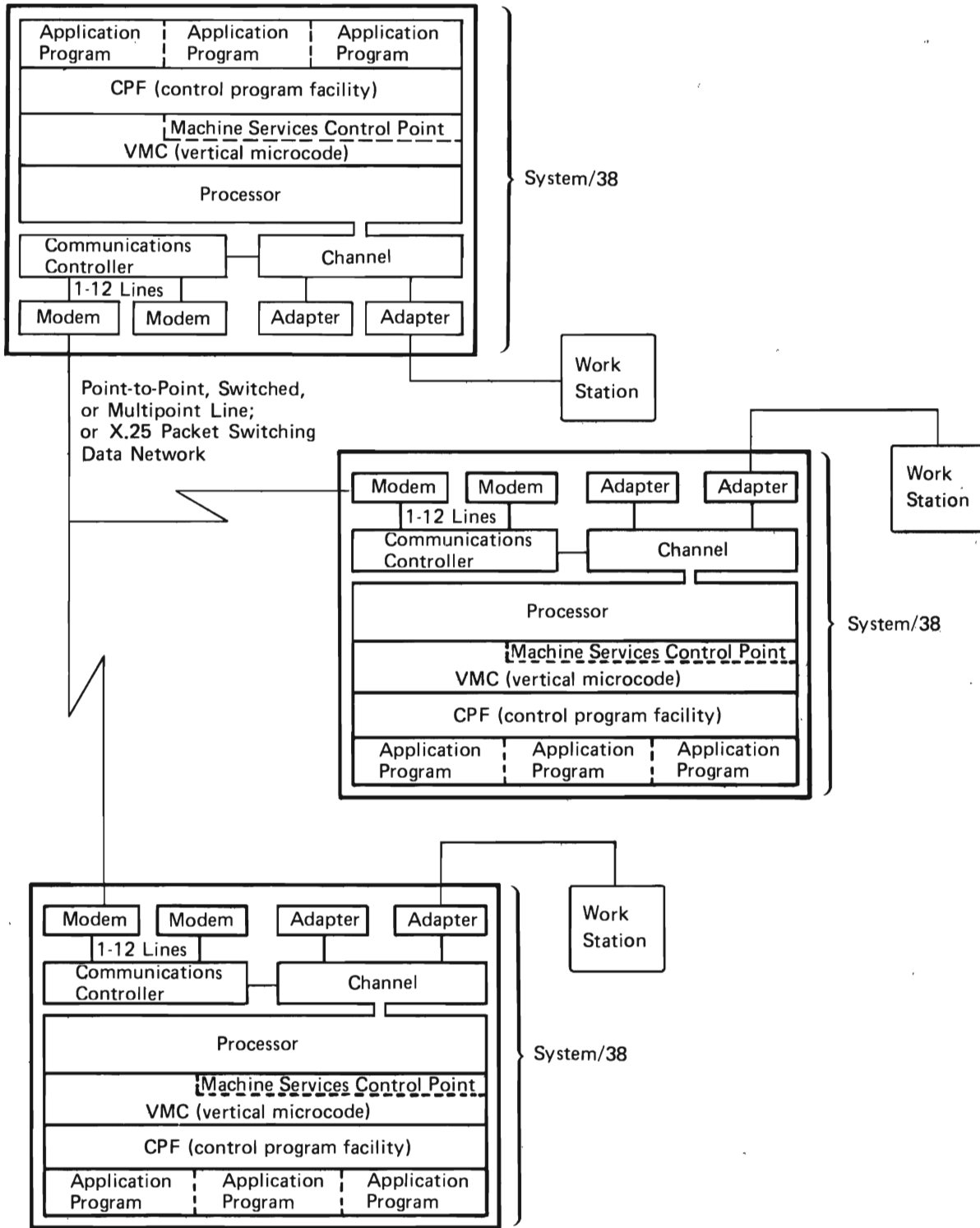
System to System Attachment (for DHCF)



¹These system objects in the System/38 programming support represent the corresponding physical elements in the I/O network.

Figure 6-1 (Part 10 of 11). Input/Output Network Components

System to System Attachment (SNA)



System/38 advanced program-to-program communications (APPC) (LU 6.2) peer support allows multiple systems to be connected on a multipoint SDLC link. The System/38 display station pass-through function uses APPC to logically connect (through intermediate nodes) a display station to a remote System/38.

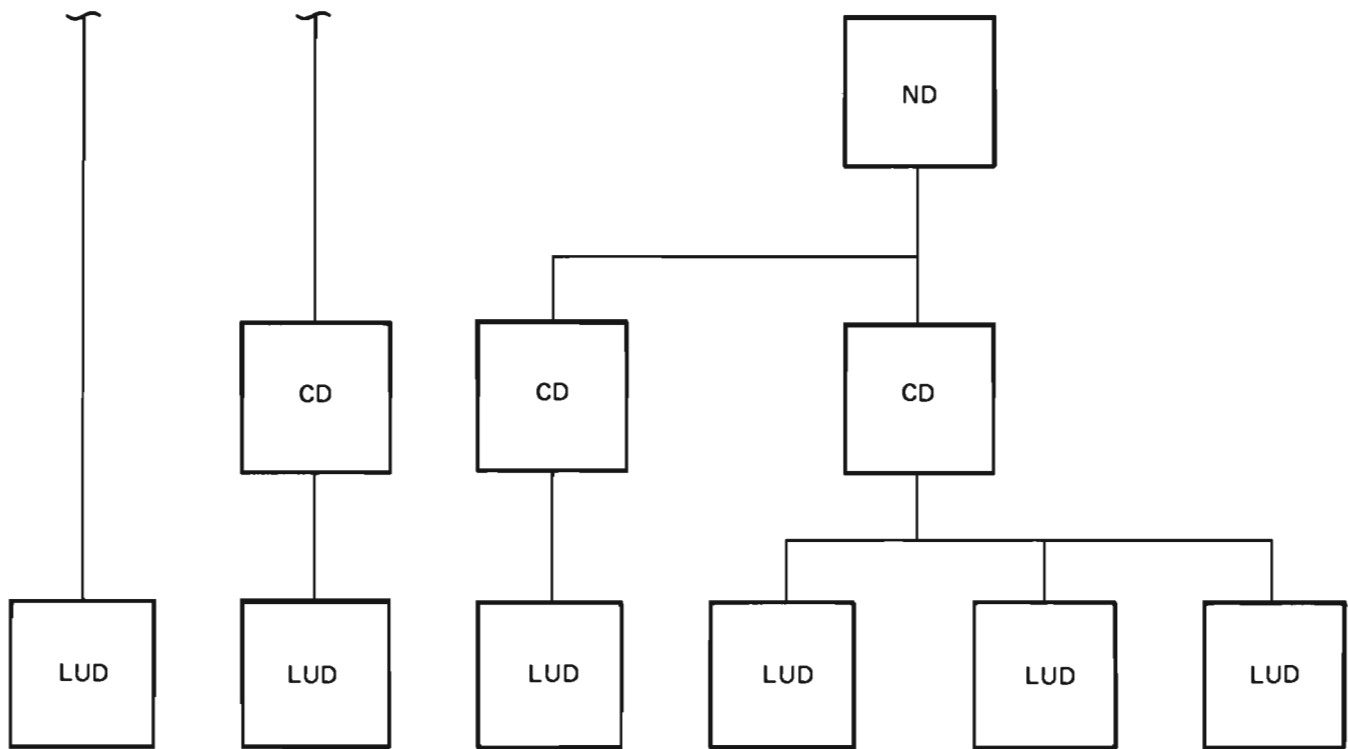
Figure 6-1 (Part 11 of 11). Input/Output Network Components

CONFIGURATIONS AND STATES OF SOURCE/SINK OBJECTS

Configurations


Source/Sink Object Subtypes

The three source/sink object types represent physical elements within a typical system. The following illustration shows the ordered relationship among the three object types.



The linkage of one object type to another object type is in the order shown in the previous illustration. Multiple CDs can attach to NDs. Multiple LUDs, in turn, can attach to each CD, but the order of object attachment is preserved because the LUD is always the ultimate end object.


Source/sink object subtypes exist because all object types (ND, CD, and LUD) are not required for every device configuration. For example, if a configuration does not require a description of any controller or port characteristics, then an LUD can exist without the support of an associated CD and ND. Similarly, a CD can exist without the support of an ND.



The source/sink objects and object subtypes must be arranged in a topological structure that shows the relationship among the objects. This structure must represent a valid configuration of I/O devices and show whether objects exist in support of these devices. This structure is defined by a set of system object pointers contained within each of the objects.

Forward and Backward System Pointers

Each system object can have a forward pointer or a list of backward pointers or both. A forward pointer always points to an object higher in the structure; a backward pointer always points to objects lower in the structure. For example, in a CD object a forward pointer points to the ND to which this CD is attached. The backward pointer list in the CD points to all the LUDs that are attached to this CD. Forward and backward pointers do not exist for the respective top and bottom objects in the structure. Depending on the source/sink object subtypes, there are several types of forward-backward pointer combinations allowed.



Configurations Defined

The object type and object subtype definitions provide several possible configurations of source/sink objects. However, based on their hardware characteristics, the system supports the configurations and their respective pointer arrangements as shown in Figure 6-2.

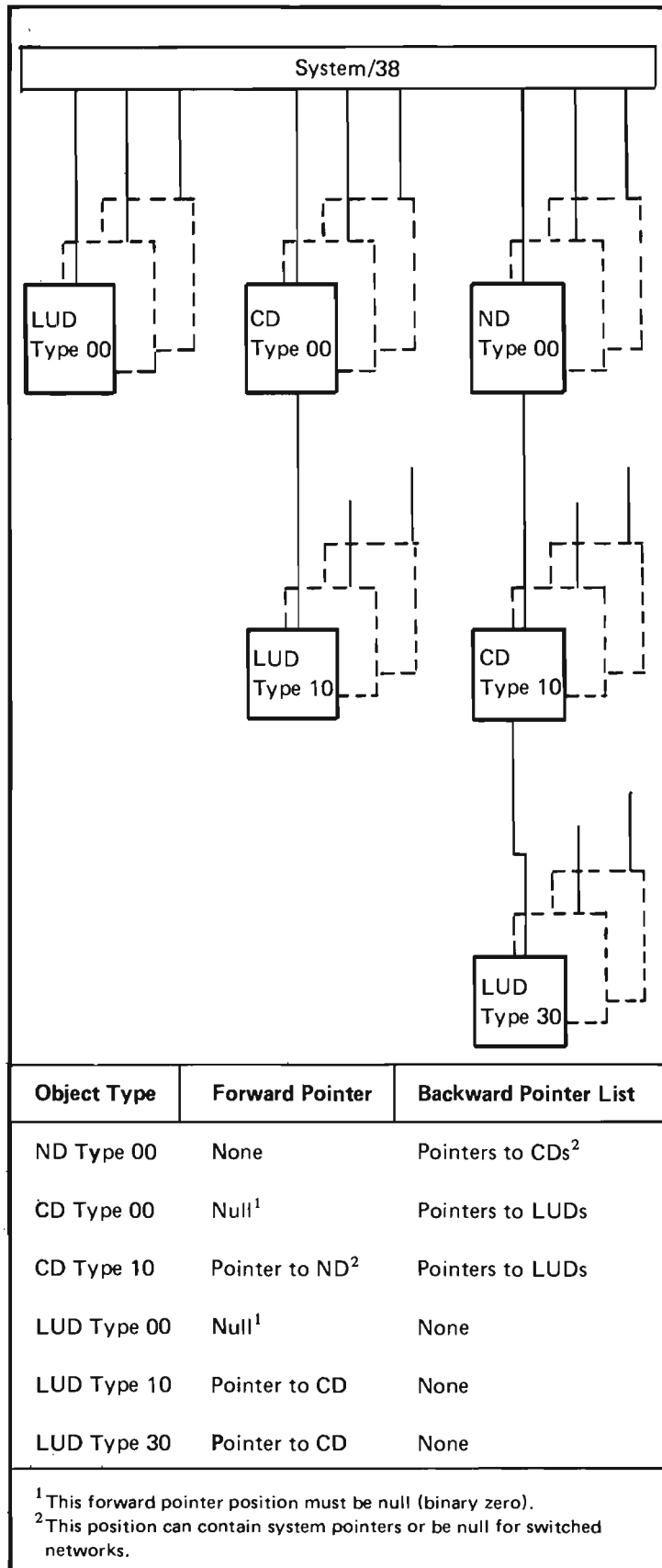


Figure 6-2. System Configuration and Pointer Arrangement

CONFIGURATION INFORMATION

The following paragraphs describe other aspects that are necessary to completely define the source/sink configuration facilities.

Machine Configuration Record

Each system has a machine configuration record that contains the internal configuration of the system. This configuration record provides the system with sufficient integrity to allow only the creation of source/sink objects for which system hardware and internal system support were physically installed. The content of the machine configuration record provides source/sink object integrity for the following conditions:

Condition	Conditions Checked
Creation of all LUD, CD, and ND objects	Internal system support must exist.
Creation of all ND objects	Internal system hardware must exist.
Creation of all CD type 00 and LUD type 00 objects	Devices must be attached to system.
Creation of CD type 10 objects and LUD types 10, and 30	No additional checking is done. Actual physical existence is not confirmed.

Materialize Machine Configuration Record

The Materialize Machine Attributes instruction can be used to materialize the machine configuration record. For information about the machine configuration record, refer to the *System/38 Functional Reference Manual*.

Object Modification Limitations

The source/sink Modify instructions do not support modification of any elements within the LUD, CD, and ND objects that correspond to physical hardware changes to devices, controllers, I/O ports, or physical installation of internal system support for this hardware.

If physical configuration changes are made to the hardware, then the source/sink Destroy and Create instructions are used first to destroy the appropriate source/sink object, and then to create a new object that includes the corresponding physical I/O configuration change.

SWITCHED NETWORK CONSIDERATIONS

All configuration information and object linkage mechanisms previously discussed were indicated as static information because the linkages established during object creation time do not change during the use of the objects. In the case of switched communications networks, these linkages must be dynamic because the hardware connections cannot be uniquely defined except for the duration of time when a switched network connection is active. A separate group of dynamic pointers, which are called switched forward and backward system pointers, are defined for this purpose.

Switched Forward and Backward Pointers

Generally, each station in a switched public telephone network or packet switched data network can be connected through different ports on a system (each station can call in on a different telephone number); conversely, each port can be connected to many different stations at different times. Consequently, the ND and CD objects, which are created to represent these ports and stations, cannot have their respective forward and backward system pointers uniquely established. Switched forward and backward pointers, therefore, are dynamically assigned and inserted into the ND and CD objects at the time that switched connections are actually made.

When a switched connection is made, an event is signaled. The CD contact event is a result of a previous Modify CD instruction (vary on) for dial in operations or a result of a Modify CD instruction (dial out) for dial out operations. By this time, the forward and backward pointers have been inserted into the objects. When a Modify CD instruction (abandon connection) is executed, the switched connection is dropped, and the pointers are deleted from the objects.

In an X.25 packet switching data network (PSDN), the procedures are similar but handled within the network on switched virtual circuit connections.

Network Description Candidate List

Each controller description contains a network description candidate list that is used to control switched network connections. The sequential list of ports (NDs) to which each station can be connected is defined by the user.

When switched connections are initiated from the system out to the station (called dial-out), the system makes the connection through the first available ND (ND enabled on) in the ND candidate list. When switched connections are initiated by the station on SDLC or X.25 data links (called dial-in), the candidate list within the CD for this calling station is searched to determine whether this station is allowed to be connected through the port (ND) on which the station called in. On BSC data links, the ND candidate list is not used for dial-in operations, but it is used only for dial-out operations.

Controller Description Eligibility List

A controller description eligibility list in the network description object is used (instead of the ND candidate list) to establish the switched connection for BSC dial-in switched operations. The machine searches for the appropriate CD by comparing the XID (exchange identification) in each eligible controller description with the XID received from the remote station. The search operation ends when a match is found. The appropriate CD is then connected to the ND for the line that received the incoming call.

OBJECT CONTENTS

Each object type (LUD, CD, and ND) consists of several elements. Each element can have varying data, type, and size. These elements include system pointers, lists of system pointers, and various types of scalar data. Although there are several elements that are indicated as variable in size, once created, each object is fixed in size. For example, two different LUDs for different I/O devices can be different sizes, but each LUD remains fixed in size after it is created. See Figure 6-3 for an example of the elements contained in a LUD. Note that each element within the object that can be materialized and modified is identified by an option value. This value is used in the operands of the Materialize and Modify instructions to specify those elements that are to be either materialized or modified.

Elements Contained in an LUD Template		Element Length	Sub Element Length	Materialize or Modify Option Values
Common Elements	1 Template size specification	Char(8)		
	Reserved (for all templates except those that include object header data)	Char(8)		
	2 Object header data (includes template size—Char[16])	Char(96)		1003
	3 LUD definition data	Char(16)		1007
	4 Pointer group data	Char(16)		1005
	5 Physical definition data	Char(16)		1009
	6 State/status definition	Char(16)		z001
Specific Elements	Session definition data	Char(32)		z002
	Load/dump definition data	Char(16)		z004
	Specific characteristics	Char (y + 2)		1012
	Retry value sets	Char (6y + 2)		z008
	Error threshold sets	Char (8y + 2)		z010
	Device-specific contents	Char (y + 4)		z020
<p>y = Variable length of an element. z = Option value control digit. Valid values are: z = 1 Materialize this individual element. z = 4 Materialize or modify this element as part of a group of modifiable elements.</p>				

Figure 6-3. Elements Contained in the LUD Template

Common Elements in LUD, CD, ND

The common elements in the LUD, CD, and ND are as follows (see Figure 6-3):

1 Template Size Specification

Template size specification contains the number of bytes of data supplied in the template for whatever operation (create, modify, materialize) is to be performed, and the number of bytes available to be materialized (used as an output field for Materialize instructions). Every input template for any Create, Modify, or Materialize instruction must contain these 8 bytes of character data as the first 8 bytes within the data area of that template.

2 Object Header Data (includes template size)

The object header data entry contains information common to all objects.

3 LUD Definition Data

LUD, CD, and ND type—This element defines the source/sink object subtype. It indicates how each object is adapted into an I/O configuration.

LUD, CD, and ND identification—Specifies the symbolic name of the object.

4 Pointer Group Data

Forward and backward object pointers describe the I/O configuration by defining how the source/sink objects relate to one another.

Switched connection forward and backward object pointers are used only for switched networks. These pointers are used in a dynamic operation to relate the source/sink objects to one another.

5 Physical Definition Data

Each object contains this address field to uniquely correlate the source/sink object with the physical hardware it represents.

6 State/Status Definition

This element presents the status or operational state of the object when viewed through a Materialize instruction. This element, when included in a Modify instruction, also serves as a command field to indicate a requested change of state.

Specific Elements in LUD, CD, and ND

Additional elements that are unique to each object type are not further defined here. These unique elements are defined in the *System/38 Functional Reference Manual*. Each element listed in Figure 6-3 is present in all objects of that type and subtype even though the contents are defined specifically for each device. For example, all LUDs have an element called session definition data even though some devices may not require this data. In this case, their respective LUDs have a null entry for this element.

Object States

Each source/sink object contains an element called the state change/status field. When the object is materialized, this field becomes a status indicator to indicate the current operational state of that source/sink object (see the following example). When the object is modified, this field becomes a state change request (command) field that allows the user to change the operating state of that source/sink object. There is not a one-to-one correspondence between the bit assignments in the Modify and Materialize instructions for each object due to the fundamental difference between status and commands.

The following is an example of a state change/status field for an LUD object.

State Change/Status Field							
Materialize (State/Status)			Modify (Commands)				
Byte(s)	Bit(s)	Meaning	Byte(s)	Bit(s)	Meaning		
0	0-6	State/Status	0	0-6	Commands		
		Reserved			Reserved		
1	7	Active session state	1	7	Activate session		
	State/Status			Commands			
	0	Suspended session state		0	Suspend session		
	1	Quiesced session state		1	Quiesce session		
	2	Reset session state		2	Reset session		
	3	Varied on/no session state		3	De-activate session		
	4	Vary on pending state		4	Vary on		
	5	Reserved		5	Vary off		
2	6	Power on/vary off state	2	6	Power on		
		7			Power off state	7	Power off
		State/Status			Commands		
		0			Diagnostic state	0	Set diagnostic mode
3-5	1	Diagnostic active indicator	3-5	1	Reset diagnostic mode		
		2-7			Reserved	2-7	Reserved
Reserved		Reserved					
0	0	Recovery/resource activation definition	0	0-1	Recovery/resource activation definition		
		0			Inoperative pending state	0-1	Reserved
		1			Normal pending state	2	Cancel
		2			Normal cancel state	3	Continue
		3			Normal continue state	4-7	Reserved
		4			Normal activation pending state	1	Reserved
		5-6			Reserved		
		7			Normal active state		
Reserved							
1							

Each object has a set of state change rules to define which commands are allowed in each respective state of that object. These rules are described under *Source/Sink Management in the System/38 Functional Reference Manual*.

Object Control States

Each object can exist in several operational states. These states are arranged in a sequence with the states higher in the sequence usually associated with increasing activity at the device itself. The states lower in this sequence are called control states because they are synonymous with control of, management of, or authorization of the use of these devices. Examples of these states are power control states, vary off-vary on states, and diagnostic states.

Object Usage States

The states higher in the state sequence are called usage states because they are synonymous with how the device is generally used. These states are sometimes controlled directly by instructions operating on that object. For example, session states (active session) in the LUD object result from a Modify LUD (activate session) instruction execution. At other times, session states are controlled indirectly through operations on related objects, such as the network active state in the ND. This state can result from activity occurring on CD objects associated with that ND.

Recovery/Resource Activation State

The recovery/resource activation state indicates the status of the communications link. This status area indicates whether error recover is necessary. Recovery/resource activation state changes normally occur asynchronous to any modify instruction. For example, the CD object recovery/resource activation state changes to inoperative pending when a line failure occurs. However, the recovery/resource activation state sometimes changes because an instruction was issued. For example, the MODCD (continue) instruction modifies the recovery/resource activation state to normal continue and starts error recovery after a failure. Note that the recovery/resource activation state is independent of the object state.

Related Instructions

The source/sink instructions associated with object state changes are Modify ND, Modify CD, Modify LUD and Request I/O. All state instruction accomplished through use of the Modify instructions. The Request I/O instruction is dependent on the operational state of the LUD object because this instruction can only be issued to an LUD that is in the active session state.

States of the LUD

The sequence of states maintained for the LUD object is as follows.

- Usage states:
 - Active session state
 - Suspended session state
 - Quiesced session state
 - Reset session state
- Control states:
 - Varied on/no session state
 - Vary on pending state
 - Varied off/power on state
 - Power off state
 - Diagnostic state

The progression through these states for normal operation is from power off, to varied off/power on, to varied on/no session, to active session, and back down in reverse order.

Suspended, quiesced, and reset session states are a group of inactive session states used for deferring the operation or doing error recovery for I/O requests within a session.

The vary on pending state exists only for devices in communications environments. It indicates that the device has been logically varied on by an MODLUD (vary on) instruction but that physical contact has not yet been established because dial in or dial out connections have not yet occurred or because station contact has not been made at the CD associated with this LUD.

Diagnostic state is used in conjunction with the control states. This state indicates that the LUD is not available for normal use because it is dedicated to the maintenance functions within the system.

Recovery/Resource Activation State of the LUD

The recovery/resource activation states defined for the LUD object are the following:

- Normal active state
- Normal activation pending state
- Normal continue state
- Normal pending state
- Normal cancel state
- Inoperative pending state

Normal active, normal activation pending, and normal continue states indicate that the LUD can operate normally. No error recovery is necessary.

The normal pending state indicates that one of the following conditions is true:

- The Request Disconnect or Disconnect SDLC command was received from the far end.
- For LU1, the MSCP-to-LU session is not active.
- For a work station, the device is not available (powered off).

The normal cancel state indicates an MODND, MODCD, or MODLUD (cancel) instruction was issued to suspend error recovery.

The inoperative pending state indicates a failure has occurred, and error recovery is necessary.

States of the CD

The sequence of states maintained for the CD object is as follows.

- Usage states:
 - Controller active–LUDs in session state
 - Controller active–varied on LUDs state
- Control states:
 - Varied on state
 - Dialing out state
 - Vary on pending (with LUDs pending) state
 - Vary on pending state
 - Varied off state
 - Diagnostic state

The normal progression through the states is from varied off to varied on, to controller active (varied on LUDs), to controller active (LUDs in session), and back in reverse order. For the CD object, the control state changes are made through Modify CD instructions, but the usage state changes occur as a result of MODLUD instructions to vary on LUDs or to activate sessions on LUDs associated with this CD.

Controller active (LUDs in session) state indicates that one or more LUDs attached to this CD are in a session state.

Controller active (varied on LUDs) state indicates that one or more LUDs attached to this CD are in a varied on state, but no LUDs are in a session state.

Dialing out state occurs only for CDs in switched communications environments and only for switched dial-out operations or X.25 switched virtual circuit operations. It indicates that the MODCD (dial out) command was issued and dial connections are in process but not yet completed, or that an X.25 CALL REQUEST packet has been sent but processing has not completed. (A CD event is raised when the connection is completed.)

Vary on pending states exist only in the communications environment and are similar to those described for the LUD object.

Diagnostic state is used (as in the LUD) to indicate that maintenance functions have made the controller unavailable for normal use.

Recovery/Resource Activation State of the CD

The recovery/resource activation states defined for the CD object are the following:

- Normal active state
- Normal activation pending state
- Normal continue state
- Normal pending state
- Normal cancel state
- Inoperative pending state

Normal active, normal activation pending, and normal continue states indicate that the CD can operate normally. No error recovery is necessary.

The normal pending state indicates the Request Disconnect or Disconnect SDLC command was received from the far end. For X.25, the equivalent logical link command was received from the far end.

The normal cancel state indicates an MODND or MODCD (cancel) instruction was issued to suspend error recovery.

The inoperative pending state indicates a failure has occurred, and error recovery is necessary.

States of the ND

The states for the network description object are as follows.

- Usage states:
 - Network active state
 - Manual dial start state
 - Manual answer start state
 - Manual answer state
 - Dial pending state
- Control states:
 - Switched enable state
 - Varied on state
 - Varied off state
 - Diagnostic state

The progression through the states of an ND depends on whether the ND is:

- Nonswitched (SDLC or BSC)
- Switched dial out (SDLC or BSC)
 - Manual dial
 - Auto dial
- Switched answer (SDLC or BSC)
 - Manual answer
 - Auto answer
- X.25

The ND for a particular line must first be explicitly varied on (via the Modify Network Description instruction) before a CD is varied on; otherwise, an exception is signaled.

The normal progression through the states for a nonswitched ND is from varied off to varied on, and then to network active. The network active state means that one or more CDs attached to this ND are either in a varied on pending state or in a higher state.

The normal progression through the states for a switched dial out ND (manual dial) is from varied off to varied on, to switched enable, to dial pending, to manual dial start, and then to network active.

The normal progression through the states for a switched dial out ND (auto dial) is from varied off to varied on, to switched enable, to dial pending, and then to network active.

The normal progression through the states for a switched answer ND (manual answer) is from varied off to varied on, to switched enable, to manual answer, to manual answer start, and then to network active.

The normal progression through the states for a switched answer ND (auto answer) is from varied off to varied on, to switched enable, and then to network active.

The normal progression through the states for an X.25 ND is from varied off to varied on, to dial pending (only if a switched virtual circuit call out is attempted), to network active.

Switched enable state occurs only for NDs in switched communications environments. It indicates that the MODND (enable) command was issued to initialize the communication line in preparation for dialing out or answer operations. Switched enable state does not occur for an X.25 ND.

Dial pending state occurs only for NDs in switched communications environments and only for switched dial-out operations and X.25 switched virtual circuit call-out operations. It indicates that an MODCD (dial out) command was issued and the related line was selected for establishing the dial connection to the desired station.

Manual dial start state occurs only for NDs in switched communications environments and only for switched manual dial-out operations. It indicates that the MODND (manual start data) command was issued after the manual dialing was completed on the data set. Manual dial start state does not occur for an X.25 ND.

Manual answer state occurs only for NDs in switched communications environments and only for switched manual answer operations. It indicates that the MODND (manual answer) command was issued after the data set was manually answered by the system operator. Manual answer state does not occur for an X.25 ND.

Manual answer start state occurs only for NDs in switched communications environments and only for switched manual answer operations. It indicates that the MODND (manual start data) command was issued after the data set was manually answered, the MODND (manual answer) command was issued, and the data set was set into data mode. This operation then completes the manual answer sequence. Manual answer start state does not occur for an X.25 ND.

Network active state (switched networks) is the conclusion of the machine operation after the various commands have been issued and the various functions performed.

Recovery/Resource Activation State of the ND

The recovery/resource activation states defined for the ND object are the following:

- Normal active state
- Normal continue state
- Normal cancel state
- Inoperative pending state

The normal active and normal continue states indicate that the ND can operate normally. No error recovery is necessary.

The normal cancel state indicates an MODND (cancel) instruction was issued to suspend error recovery.

The inoperative pending state indicates a line failure has occurred, and error recovery is necessary.

Vary On/Off Sequence

The required sequence to vary on source/sink objects is:

1. ND
2. CD
3. LUD

The required sequence to vary off source/sink objects is:

1. LUD
2. CD
3. ND

SOURCE/SINK INSTRUCTIONS

Each of the three source/sink object types is supported by four System/38 instructions: Create, Destroy, Materialize, and Modify. These instructions configure and manage the network of I/O devices on the system. One additional instruction, Request I/O, is the instruction through which device functions are actually requested.

Instruction Usage

The Create and Destroy instructions define the I/O configuration of each system. The source/sink objects that support the I/O configuration can be created as part of the system during manufacture and also created when the system is installed. When additional or replacement hardware is installed, these objects may be destroyed and new objects created. Each Create instruction ensures that the actual hardware and support is installed on the system before allowing the source/sink object to be created.

Each system defines a base set of source/sink objects that were created when the system was built. This procedure facilitates system installation and system specialization. Additional information about the objects that are created when the system is built is contained in the *System/38 Functional Reference Manual*.

Modify instructions are used primarily for managing the network and device facilities. Some typical functions accomplished through use of the Modify instruction are: controlling which devices can be used and which devices are unavailable, or which stations are allowed to call in on which ports.

The Request I/O and Modify LUD instructions are used to control I/O devices. The Modify LUD instruction controls the operational state of the LUD and also controls the device usage parameters. The Request I/O instruction requests device functions. The request I/O functions are, in general, device dependent. Each system maintains a record of its installed I/O devices, its configuration, and the corresponding device operational commands and capabilities as supported through the Request I/O instruction. For more information about the Request I/O instruction, refer to *Source/Sink Configuration* in the *System/38 Functional Reference Manual*.

The Materialize instructions are used to determine the I/O configuration and the operational states of the system hardware components.

The Request Path Operation instruction can be used to establish an internal path between objects and functions supported within the machine. It can be used to request that a path be established between two logical unit descriptions specified in the template input to the instruction. It can also be used to request that a path be established between the load dump support provided within the machine to a dump space specified in the template input to the instruction. Additionally, load or dump operations utilizing this path can be initiated through use of the instruction.

The template specifically describes the transaction and contains the data involved with the transaction. The template also specifies the queue (request path operation response queue) to which an asynchronous message (called a feedback record) is to be sent by the machine when the transaction requested by this instruction is complete.

Exclusive Locks on Source/Sink Objects

The integrity of source/sink objects and the integrity of the operations involving these objects when they are shared by two or more processes is ensured by system object locks. However, when source/sink objects are locked exclusive, not all elements within these objects necessarily remain unchanged for the duration of the lock. For example, the state change/status elements of all LUD, CD, and ND objects are subject to being changed independently of any process actions taking place. These elements can be changed (even though they are locked) because of the usage of source/sink objects. The changes can occur due to actions within the machine not directly associated with any process. For example, an ND object representing a switched line in answer mode will change to active state if an incoming call occurs and the call is not directly associated with any user operations on this ND object.

Events

Asynchronous events are used to indicate conditions that occur in the network independently from (although related to) the source/sink instructions. Asynchronous events exist for all three object types: LUD, CD, and ND.

Create/Destroy Instructions—Hierarchy Rules

Create Instructions

Uniqueness: The Create Logical Unit Description (CRTLUD), Create Controller Description (CRTCD), and Create Network Description (CRTND) instructions create the source/sink objects that define and control the hardware I/O components within the system and the network of devices.

These objects must be created to be unique in the following ways:

Unit (Device) Type, Model, and Source/Sink Object Subtype: Each System/38 model supports certain unit types and the corresponding source/sink object subtypes for these units. An exception is signaled if the unit type, model number, and source/sink object subtype do not correspond. The data related to these unit types and object subtypes is contained in the *System/38 Functional Reference Manual*.

Corresponding Physical Hardware: When a source/sink object is created, the machine configuration record within the system is used to verify that physical hardware and system support capabilities exist for the object. If the physical hardware and the system support capabilities cannot be verified (based on physical address, unit type, model number, and other elements), an exception is signaled.

Note: The physical hardware that is external to the system does not have an entry in the machine configuration record and cannot be confirmed.

Unique Physical Address within Each Object Type: No two source/sink objects of the same type (for example, two LUDs) can use the same physical address on the system. Because these same unique addresses are also assigned to physical hardware components, the assignments allowed to the source/sink objects are likewise checked for uniqueness. In the case of ND and LUD objects, this uniqueness checking is based only on the physical address field within each object. In the case of CD objects for SDLC communications controllers, this uniqueness is based on the XID (exchange identification) field (work station, APPC) and SSCPID field (LU1).

Regarding CD objects for X.25 communications controllers, the logical channel ID is used for this uniqueness checking for a permanent virtual circuit. For a switched virtual circuit CD, the remote network address and network connection password are used for uniqueness checking.

Multiple objects (NDs, CDs, and LUDs) with duplicate physical addresses can be created for communications networks. For example, you can create up to 10 NDs with operational unit 20 as the physical address (System/38 communications line 1). These multiple objects are used uniquely at vary on time because only one set of objects for each physical configuration can be active at one time.

Physical address checking for BSC controllers is bypassed because the CD objects can represent a generic class of BSC stations as well as one unique station.

Destroy Instructions

The three unique destroy instructions—Destroy Logical Unit Description (DESLUD), Destroy Controller Description (DESCD), and Destroy Network Description (DESND)—are used to destroy the source/sink objects. When a source/sink object is destroyed, the related source/sink objects referenced by the forward or backward object pointers are searched. If these related objects exist, their corresponding backward or forward object pointers to this object are nullified.

Configuration Hierarchy Rules

Creation and/or destruction of source/sink objects occurs primarily as part of system specialization during the time a system is built and installed. Source/sink objects must also be created and destroyed to reflect hardware changes and additions that occur during the lifetime of a system.

Object creation and destruction should occur in a preferred sequence to minimize the user's involvement in defining the forward and backward chaining of source/sink objects. The preferred sequence is to create NDs, CDs, and then LUDs in that order within each related set and to destroy them in reverse order—LUDs, CDs, and lastly NDs.

When the ND, CD, and LUD objects are created in the preferred sequence, the user need only supply the forward pointers within each create template. No backward pointer lists need be supplied because the system builds these lists as part of creating the subsequent objects on the chain.

For those source/sink objects that are created out of sequence, the user has the additional obligation of supplying the list of backward pointers so that the system has enough information to complete the chaining of the objects and can insert the corresponding forward pointers in these related objects.

When the objects are destroyed in reverse order, the system methodically removes the entries from the backward lists and thus minimizes the number of source/sink objects that are left in a disconnected, unusable state. Although the preferred sequence is recommended, it is not mandatory.

The secondary requirement for configuration changes is the need to not be forced into destroying all lower objects within an object chain in order to destroy the higher object in the chain. For example, if new hardware features or capabilities are added to a controller and a CD is being destroyed so that a new CD can be created in its place to reflect these additions, then it is not necessary to destroy all attached LUDs before destroying this CD. You must, however, preserve the forward pointer and the list of backward pointers to the LUDs, destroy the original CD, and recreate the new CD supplying the forward pointer and the list of backward pointers preserved from the original CD.

Source/sink objects that do not have proper forward chaining can exist in the system even though objects existing in this state are not usable until their forward chaining is properly established. These objects can be materialized or have their parameters modified, but a Modify instruction issued to change the state (status field) of that object results in a configuration invalid exception. The exception occurs until such time as that object's forward and backward chaining is properly established. The chaining is properly established when the required objects are created.

Materialize Instructions

Three unique materialize instructions—Materialize Logical Unit Description (MATLUD), Materialize Controller Description (MATCD), and Materialize Network Description (MATND)—are used to materialize source/sink objects.

Each Materialize instruction has a 2-byte character scalar operand that identifies the element or group of elements to be materialized. Each element within the object has an associated option value that can be inserted into this operand to materialize that element. Other option values allow multiple elements as well as the entire group of elements to be materialized.

Refer to Figure 6-4 and note the materialize option value assigned to each element. The digit 1 in the leftmost position of the option value indicates that only this single element can be materialized on one instruction. The Z character (which can have a value of either 1 or 4) in the leftmost position indicates that a single element (1) or a group of elements (4) can be materialized.

A group of elements can be materialized on one instruction by specifying an option value that is the result of performing a logical OR on the option values of the desired elements. (The digit 4 must also be substituted for the Z.) For example, an option value of hex 4018 specifies that the retry value sets (hex 4008) and the error threshold sets (hex 4010) elements will be materialized on one Materialize instruction.

Note: All elements are materialized on one instruction when an option value of hex 8000 is specified, except for X.25 runtime statistics.

For a complete description of the LUD, ND, and CD templates, refer to the *System/38 Functional Reference Manual, Volume 1*.

Elements Contained in the Template (LUD types 00, 10, 30) for Materialize LUD	Element Length	Sub Element Length	Materialize Option Values
Template size specification	Char(8)		
Reserved (for all materialize templates except ones including object header data)	Char(8)		
Object header data (includes template size)	Char(96)		1003
LUD definition data	Char(16)		1007
Pointer group data	Char(16)		1005
Physical definition data	Char(16)		1009
State/status definition	Char(16)		z001
• Object state		Char(6)	
<i>Byte(s)</i> <i>Bit(s)</i> <i>Meaning</i>			
0			
	0-6	Reserved	Bit(7)
	7	Active session state	Bit(1)
1		State/Status	
	0	Suspended session state	Bit(1)
	1	Quiesced session state	Bit(1)
	2	Reset session state	Bit(1)
	3	Varied on/no session state	Bit(1)
	4	Vary on pending state	Bit(1)
	5	Reserved	Bit(1)
	6	Power on/vary off state	Bit(1)
	7	Power off state	Bit(1)
2		State/Status	
	0	Diagnostic state	Bit(1)
	1	Diagnostic active indicator	Bit(1)
	2-7	Reserved	Bit(6)
3-5		Reserved	Char(3)
• Recovery/resource activation definition		Char(2)	
0			
	0	Inoperative pending state	Bit(1)
	1	Normal pending state	Bit(1)
	2	Normal cancel state	Bit(1)
	3	Normal continue state	Bit(1)
	4	Normal activation pending state	Bit(1)
	5-6	Reserved	Bit(2)
	7	Normal active state	Bit(1)
1		Reserved	Char(1)
• Reserved		Char(8)	
Session definition data	Char(32)		z002
Load/dump definition data	Char(16)		z004
Specific characteristics	Char (y + 2)		1012
Retry value sets	Char (6y + 2)		z008
Error threshold sets	Char (8y + 2)		z010
Device-specific contents	Char (y + 4)		z020
y = Variable length of an element. z = Option value control digit. Valid values are: z = 1 Materialize this individual element. z = 4 Materialize this element as part of a group of modifiable elements.			

Figure 6-4. Materialize LUD Template

Modify Instructions

The three unique modify instructions—Modify Logical Unit Description (MODLU Modify Controller Description (MODCD), and Modify Network Descriptions (MODND)—are used to modify the source/sink objects.

Each modify instruction has a 2-byte character scalar operand that identifies the element or multiple elements to be modified. Each modifiable element within the object has an associated option value that can be inserted into this operand to modify that element. Multiple elements can be modified as a group by logically ORing together those individual option values to be included in the group.

Refer to Figure 6-5 and note the option value assigned to each element. You can modify, for example, the state/status definition element by specifying the option value assigned to the element (hex 4001). You can also modify multiple elements (such as, the session definition data (hex 4002), the retry value sets (hex 4008), and the device-specific contents (hex 4020)) as a group by specifying an option value of hex 402A, which is the result of performing a logical OR operation on the option values assigned to these elements.

Elements Contained in the LUD Template (LUD types 00, 10, 30) for Modify	Element Length	Sub Element Length	Modify Option Values
Template size specification	Char(8)		
Modify time-out value	Char(8)		
State/status definition	Char(16)		4001
• Object state change		Char(6)	
Byte(s) Bit(s) Meaning			
0			
	0-6	Reserved	Bit(7)
	7	Activate session	Bit(1)
1			
	0	Suspend session	Bit(1)
	1	Quiesce session	Bit(1)
	2	Reset session	Bit(1)
	3	De-activate session	Bit(1)
	4	Vary on	Bit(1)
	5	Vary off	Bit(1)
	6	Power on	Bit(1)
	7	Power off	Bit(1)
2			
	0	Set diag mode	Bit(1)
	1	Reset diag mode	Bit(1)
	2-7	Reserved	Bit(6)
3-5		Reserved	Char(3)
• Recovery/resource activation definition		Char(2)	
0			
	0-1	Reserved	Bit(2)
	2	Normal cancel	Bit(1)
	3	Normal continue	Bit(1)
	4-7	Reserved	Bit(4)
1		Reserved	Char (1)
• Reserved		Char(8)	
Session definition data	Char(32)		4002
Load/dump definition data	Char(16)		4004
Retry value sets	Char (6y + 2)		4008
Error threshold sets	Char (8y + 2)		4010
Device-specific contents	Char (y + 4)		4020

Note: A combination of elements can be modified on the same Modify instruction by supplying a value that is the result of performing a logical OR on the modify option values of the desired elements.
y = Variable length of an element.

Figure 6-5. Modify LUD Template

Modification Sequences

The modifiable elements contained within each source/sink object are separated (based on use) into two distinct categories. The first category contains the state change/status element with all its associated subelements. This element provides the method to control the state of each source/sink object. The second category contains all other modifiable elements within an object. These elements (referred to as operational parameters) can be changed, as required, to provide different operational functions for each source/sink object.

The Modify instructions, consequently, have two basic functions to perform. First, for the state change/status element, only certain state changes or state change sequences are allowed. The Modify instructions enforce these state change transition rules.

For the second category of elements, the operational parameters can be changed only when the object is in a state that allows the change. The Modify instructions enforce the allowable state change rules for operational parameter elements. Each operational parameter has a rule to indicate in which states that parameter can be modified. It can then be modified in all states lower than this state.

Because of these two categories of modifiable elements, modify operations must be performed in a specific sequence when a Modify instruction specifies an option value for a group of elements to be modified together. If this was not done, ambiguous cases could exist as to when state change rules apply to each modifiable element. The sequence for all object types for performing a Modify instruction with a group of elements is always as follows:

1. All state change requests that cause a transition to a lower state are completed first. These changes themselves are performed in sequence from the highest state downward.
2. All requested operational parameter elements are modified in a sequence defined for each object type.
3. All state change requests that cause a transition to a higher state are performed in an upward sequence.

LUD Session State Changes

The session states in the LUD objects were previously defined as the usage states for the LUD. This concept of a session with an LUD implies usage or control of the related device. The session states control the operations that are performed by that LUD by controlling the execution of the Request I/O instruction and the disposition of outstanding Request I/O instructions during session state changes.

Activating a Session

Sessions can first exist only in the active session state. A Modify LUD instruction to activate a session causes the LUD to move from the varied on/no session state to the active session state. Only in the active session state can a Request I/O instruction be issued to this LUD. The session can be de-activated from this state by an MODLUD (de-activate) instruction, which changes the LUD to the quiesced state first and then to the varied on/no session state. The three other inactive states—suspended, quiesced, and reset—can only be entered from the active session state. These states represent various ways of controlling the activity taking place at the LUD by defining different dispositions for any request I/O operations that were issued to this session when it was an active session and that are currently outstanding to this session.

Suspending a Session

An active session that is suspended with an MODLUD (suspend) instruction causes the system to stop processing any further Request I/O operations posted to this session and to complete any request I/O activity already started in the network. When the suspend operation is completed, all previously issued request I/O operations are either completed properly and so indicated or are still suspended within the system. The session can be reactivated by an MODLUD (activate) instruction that causes the LUD to return to the active session state and processing to resume with no loss in continuity. The session can be de-activated from the suspend session state by an MODLUD (de-activate) instruction. This causes the LUD to first go to the reset state and from there to the varied on/no session state.

Quiescing a Session

An active session can be quiesced by an MODLUD (quiesce) instruction. A transition to the quiesce state causes the system to complete all previously issued request I/O operations normally before the LUD is put into a quiesced state. The quiesced session can be reactivated by an MODLUD (activate) instruction that returns the LUD to the active session state and continues normal processing. The session can be de-activated from the quiesced state by an MODLUD (de-activate) instruction. This causes the LUD to change to varied on/no session state. Before issuing an MODLUD (quiesce) instruction, a user should be aware of the particular request I/O operations that may be pending for the device in question. Otherwise, excessive time can occur before the instruction is completed. For example, if the pending operations include a long wait for an I/O operation (such as a read from keyboard command), the quiesce operation will not complete until either the wait is satisfied by the I/O device or the modify time-out interval has expired.

Resetting a Session

An active session can be reset by an MODLUD (reset) instruction that causes the system to immediately stop processing any request I/O operations in process and to complete that request I/O and all subsequent request I/O operations abnormally and immediately return them with a reset status condition indicated. A session can be reactivated by an MODLUD (activate) instruction from the reset state, but processing is neither completed nor continued for request I/O operations issued before the reset condition. The session can also be deactivated by an MODLUD (de-activate) instruction that returns the LUD to the varied on/no session state.

De-activating a Session

Sessions can be de-activated from the active session state or any of the three inactive states (suspended, quiesced, or reset). When sessions are de-activated from the active or suspended states, previous request I/O operations may still be in process within the system because the Request I/O instruction is asynchronous to the actual request to the device.

When a session is de-activated from an active session state, the session de-activation goes through a quiesced state to allow request I/O operations (if any) to complete normally before moving the LUD to a no session state. This sequence establishes the proper protocol for ending all session (or usage) activity on the LUD. The Reset Session command must be used to de-activate a session when any abnormal requirements exist.

When a session is de-activated from a suspended state, the session goes through a reset state that causes suspended request I/O operations (if any) to be terminated abnormally and so indicated in their status.

State Change Transition Rules

Figure 6-6 is an example of the state change transition rules for an LUD (logical unit description) and the bit designation within the state change field that must be specified on a Modify instruction to cause the transition to occur.

Multiple state transitions can occur on a single Modify LUD instruction by specifying the correct bits in the state change field of the LUD template. Source/sink management causes the transitions to occur in a valid sequence if the correct bits are specified. When the state transitions cannot occur in a valid sequence, an exception is signaled.

The state of the object can be determined by issuing a Materialize instruction with the state/status definition element specified (option value hex 1001). Figure 6-6 (Part 1 of 2) shows the relationship of the bits in the state change field (Materialize instruction) and the state of the object.

Additional information about the state transitions for the CD (controller description), ND (network description), and the LUD (logical unit description) is contained in the *System/38 Functional Reference Manual*.

Figure 6-6 (Part 2 of 2) shows the operational parameter elements of an LUD and the states of the object that allow the parameter to be modified. The elements and the sequence in which the modifications occur are determined by the modify option value. For example, if an option value of hex 4036 is specified, the session information, load/dump indicator, error threshold sets, and the device-specific contents operational elements are modified in the sequence as shown in the figure. (The hex 4036 option value is the result of performing a logical OR operation on hex 4002, 4004, 4010, and 4020.)

Materialize Instruction

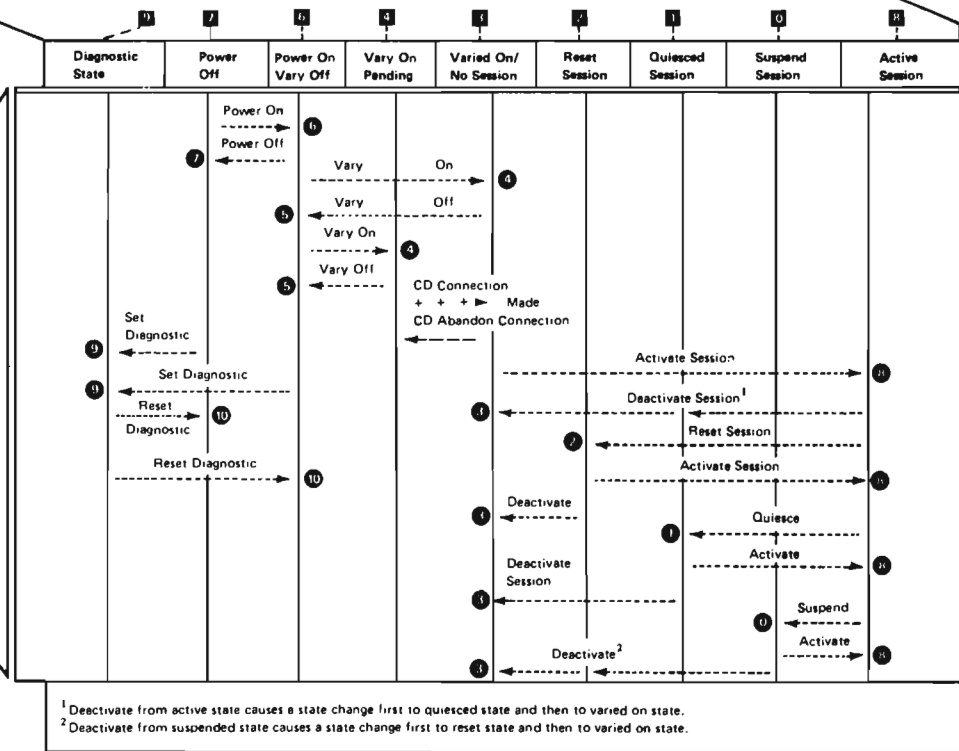
State Change/Status Field

- Byte 0 status
 - Bits 0-6 reserved
 - Bit 7 Active session state
- Byte 1 status
 - Bit 0 suspended session state
 - Bit 1 quiesced session state
 - Bit 2 reset session state
 - Bit 3 varied on/no session state
 - Bit 4 vary on pending state
 - Bit 5 reserved
 - Bit 6 power on/vary off state
 - Bit 7 power off state
- Byte 2 status
 - Bit 0 diagnostic state

Modify Instruction

State Change/Status Field

- Byte 0 commands
 - Bits 0-6 reserved
 - Bits 7 activate session
- Byte 1 commands
 - Bit 0 suspend session
 - Bit 1 quiesce session
 - Bit 2 reset session
 - Bit 3 deactivate session
 - Bit 4 vary on
 - Bit 5 vary off
 - Bit 6 power on
 - Bit 7 power off
- Byte 2 commands
 - Bit 0 set diagnostic mode
 - Bit 1 reset diagnostic mode



Meaning of Symbols:

- > State transitions due to MOOLUD operations
- > State transitions due to MODCD on the related CD
- + + + +> State transitions by system on behalf of MODCD

¹ Deactivate from active state causes a state change first to quiesced state and then to varied on state.
² Deactivate from suspended state causes a state change first to reset state and then to varied on state.

Figure 6-6 (Part 1 of 2). Summary of State Change Transition Rules (LUD)

		Allowable States for Modifying LUD Elements								
Operational Element Modify Sequence	Hex Value	Diagnostic Mode	Power Off	Power On Vary Off	Vary On Pending	Varied On/No Sessio	Reset Sessio	Quiesce Session	Suspend Session	Active
1. Session information	4002	No	Yes	Yes	No	No	No	No	No	No
2. Load/dump indicator	4004	No	Yes	Yes	Yes	Yes	Yes ¹	Yes ¹	Yes/No ¹	No
3. Retry value sets	4008	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4. Error threshold	4010	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
5. Device-specific contents	4020	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

¹Can be modified only if the load/dump device can be interrupted. Refer to the *System/38 Functional Reference Manual* for additional information about the load/dump indicator.

Figure 6-6 (Part 2 of 2). Summary of State Change Transition Rules (LUD)

Nonsupported Session State Changes

No other changes between session states are supported. For example, attempting to move from the suspended state to the quiesced state is not a valid transition. An MODLUD instruction that specifies such a transition causes an exception to occur.

Modify Time-Out Values

The modify time-out value field is used to specify the desired length of time (in standard time units) that the machine should allow for the modification operation to complete. If the modify operation does not complete within the specified time, the operation is terminated, and the partial system object damage exception is signaled. Error recovery procedures must be invoked to perform any shutdown or cleanup operations if this exception occurs.

If a time-out value of zero is specified in the modify template, a default value is used. Any nonzero time-out value supplied must fall within the time-out limits. This time-out value should not be construed as a maximum length of execution time for the Modify instruction. The time-out is only used internally to time some arbitrary portion of the operation to prevent the Modify instruction from never completing. Time-out will not occur in less than the specified time-out value. However, a valid execution may last much longer than the time-out value when several elements are included in one Modify instruction because each of the element operations is timed separately.

The minimum, maximum, and default time-out values are contained in the *System/38 Functional Reference Manual*.

Request I/O Instruction

Request I/O Functions

The Request I/O instruction schedules work to the devices within the source/sink network. The fundamental types of work achieved by the Request I/O instruction are as follows:

- The normal request I/O operation allows a unit of work to be scheduled to an I/O device in the network. This device must be represented by an LUD object in the active session state and also be the target of the request I/O operation.
- The Request I/O instruction for the load/dump operation is similar to a normal request I/O in that it involves an LUD and its associated device as the target object. But it differs in the type of work it does within this active session on that LUD. The LD (load/dump) function enables the user to save (backup) certain permanent objects by dumping these objects to an LD media (such as diskettes) and then restoring or loading these objects as needed.
- The Request I/O instruction for an MSCP (machine services control point) operation allows a request to be sent directly to the MSCP to provide network support services on behalf of any LUD or CD within the system.
- The Request I/O instruction for a service operation allows a request to be sent to the service (maintenance) components within the system to initiate various service functions on behalf of the I/O device.

The Request I/O instruction executes asynchronously to the actual performance of the work requested by that instruction. At the time the Request I/O instruction has completed, the work has been scheduled to the system; however, no direct relationship can be established concerning when the work is actually completed. When the work is completed, the system posts a feedback record to the request I/O response queue. The user can then obtain this message by issuing a Dequeue instruction against that queue.

The general sequence used to perform work on an I/O device is as follows:

1. Create a request I/O response queue.
2. Lock the LUD object (optional) for the intended device.
3. Modify the LUD to establish an active session.
4. Issue a Request I/O instruction that specifies (via a source/sink request): the LUD to be used, the request I/O response queue on which to post the feedback record results, the type of work to be done (normal, load/dump, service, or MSCP), the unit of work to be done (command or commands), and the associated data areas for this work.
5. Dequeue from the request I/O response queue to determine the disposition of the work requested (successfully accomplished or unsuccessful for specified reasons).
6. Reissue Request I/O and Dequeue instruction pairs (4 and 5) for the remaining work to be completed.
7. Modify the LUD to de-activate the session.
8. Unlock the LUD to make the device available for the next user.
9. Destroy the request I/O response queue (if no longer needed).

Variations to this sequence are possible within the checks that are enforced by the Request I/O instruction. For Request I/O instructions issued to the service processor, an LUD object is not directly associated with the Request I/O; therefore, the Lock LUD, Modify LUD, and Unlock LUD instructions (steps 2, 7, 8) are not necessary.

The Request I/O instruction relates to the request I/O response queue in the same way the Enqueue instruction relates to any other queue. The Request I/O instruction does, indeed, take on many of the characteristics of the Enqueue instruction. But the Request I/O instruction is also fundamentally different because it represents an indirect path from the user out through the I/O device and back before a message is enqueued to the request I/O response queue.

Request I/O Syntax

The Request I/O instruction consists of an operation code with one operand:

Operation Code	Operand 1
REQIO	Space pointer to an SSR

Operand 1 references a template called the SSR (source/sink request). The SSR contains all the information associated with each request I/O operation. The feedback record, which is a message enqueued to the request I/O response queue when the actual requested operation is completed, contains a pointer to this same SSR. This pointer can be used to obtain all the information from the SSR associated with the request.

The target object of a request I/O operation can be an LUD for normal and load/dump requests, an LUD or CD for MSCP (machine services control point) network services requests, or an LUD, a CD, an ND, or a null pointer for service function requests. For more information about MSCP, refer to *Machine Services Control Point* (MSCP) later in this chapter. For normal request I/O, load/dump, and MSCP requests, a system pointer to the LUD or CD is the first element in the SSR. The location of the MSCP is implicitly known to the machine.

Contents of Source/Sink Request

Source/Sink Request: The SSR consists of pointers and data that contain the following:

- Data
 - Template size specification: This entry defines the standard template header data. The size of the template field must indicate a sufficient number of bytes to contain all entries in the SSR.
 - Request I/O time-out: This field is used to specify the desired length of time (in standard time units for the system model being used) that the machine should allow for a synchronous request I/O (with task switching) operation to complete.
 - Pointers
 - First pointer: System pointer to an LUD for normal and load/dump request I/O operations. System pointer to an LUD or a CD for MSCP request I/O operations. System pointer to an LUD, a CD, an ND, or a null pointer for service function request I/O operations.
 - Second pointer: System pointer to the request I/O response queue.
 - Third pointer: Space pointer to a SSD (source/sink data area).
 - Fourth pointer: This space is reserved for an optional pointer. When not used, this space is treated as reserved and must contain binary zero.
 - Data
 - Request I/O timestamp: This field is set by the machine to indicate (in standard time units for the system) the time when this request was processed. The request I/O response queue contains a standard enqueue timestamp that is also set to indicate the time of actual completion of the resulting I/O operation.
 - Request priority: This field is used to establish the priority of each Request I/O instruction relative to each other. This key is used to establish priority of the request as it is issued to the machine and as the requested operation is scheduled. Priority values are established with hex 0000 being the highest priority and hex FFFF being the lowest priority.
- Request ID: This field is used to assign any unique identification to each source/sink request. This identification enables a user to associate feedback records with the Request I/O instruction that generated them. This ID is inserted into the message area supplied for the Dequeue instruction that retrieves the feedback record from the request I/O response queue. The message area can then be inspected to associate the message with the originating Request I/O instruction. The request ID field is also used to control the signaling of the request I/O completed event. When bit 0 of this field is equal to 1, the request I/O completed event is signaled at the time the feedback message is enqueued. This indicates that the processing of this request is completed. When bit 0 of this field is equal to zero, no event is signaled.
 - Function field: This field specifies the request I/O type. Bits 0–3 define the type of request; bits 4–7 are function-dependent and are uniquely defined for each device. For more information about the function field, refer to *Source/Sink Management Instructions* in the *System/38 Functional Reference Manual*.
 - Request control field: This field is used to define request I/O control functions. For more information about the request control field, refer to *Source/Sink Management Instructions* in the *System/38 Functional Reference Manual*.

The Request I/O (continue) instruction defines the treatment for previously issued Request I/O instructions when error situations have been encountered within the machine. Certain error situations can arise that cause feedback records (which indicate the error) to be posted for those requests. This means that terminating errors have occurred; all further processing of scheduled request I/O operations was suspended, and the machine is waiting for further information on disposition of these suspended requests. If additional normal Request I/O instructions are issued during this time, they are executed (no I/O function occurs) and enqueued to the stack of suspended requests.

Request I/O (continue) instructions can be executed at this time to control certain error recovery situations. A Request I/O (continue) instruction causes the machine to restart normal operations immediately after posting a feedback record corresponding to this Request I/O (continue) instruction.

The priority field in the SSR can be used on these Request I/O (continue) instructions just as in normal Request I/O instructions for the user to establish priority. However, internally these operations are assigned a higher priority than the normal request I/O operation so that the request I/O (continue) operations always take priority over outstanding normal I/O requests.

- Key length: This field indicates the length of the request key field in this SSR. The value specified in this field must also match the key length attribute of the response queue specified in this SSR.
- Offset to key field: This field locates the request key field within the SSR. The positive value contained in this field defines a location that starts from the beginning of the SSR.
- Request key: This field is used by the machine to post the feedback record onto the response queue. Then the Dequeue instruction can use this same key value to retrieve the feedback record corresponding to this Request I/O instruction.
- Request descriptor count: This field indicates the number of request descriptor fields that follow in this SSR.
- Offset to request descriptors: This field locates the request descriptor field within the SSR. The positive value contained in this field must define a location that is either 2-byte aligned for normal, MSCP, or service requests or 16-byte aligned for load/dump requests. The offset starts from the beginning of the SSR.
- Request descriptor: The RD (request descriptor) fields are the command operations for the I/O device or the message description to be sent to a communication device. The request descriptors are either 96-byte entries for load/dump operations or 16-byte entries for all other operations. The RD contents are uniquely defined for each type of I/O device on the system.
- Offset to variable parameters: This field indicates the location within the SSR where the variable parameters (if any) have been placed. This offset is defined as a positive value offset from the beginning of the SSR and must define a 16-byte aligned location.
- Request I/O variable parameter: This variable parameter area is used by certain devices or support mechanisms to define additional data which is necessary for their support.

Source/Sink Data

The SSD (source/sink data) is a space that contains the data areas (I/O buffers) associated with the operations requested by the request descriptors within the SSR. For normal request I/O operations, the SSD is a space; however, the SSD optionally need not be present if no data is read or written. The SSD is further subdivided into elements called RIUs (request information units) that correspond to the request descriptors in the SSR (source/sink request). (The load/dump request I/O operation does not use an SSD.) The contents of the SSD are defined in the *System/38 Functional Reference Manual*.

Request I/O Response Queue and Feedback Record

The request I/O operation requires an LUD object and a request I/O response queue object. The LUD represents the device. The response queue contains the messages generated by the machine (asynchronous to the Request I/O instruction). These messages indicate the final disposition of the requested unit of work that was specified by that Request I/O instruction.

The messages on the request I/O response queue, which are retrieved by a standard Dequeue instruction, are called FBRs (feedback records).

Request I/O and Request I/O Response Queue Relationship

The following information describes the special relationship between the request I/O operation and the response queue, the special considerations when creating and using a queue as the request I/O response queue, and the format and contents of the feedback records.

A request I/O operation acts as an Enqueue instruction on a keyed queue because request I/O operations are processed in binary collating key sequence (subject to time of arrival considerations). For example, internal request I/O processing occurs on the request with the highest priority of all requests that were issued (enqueued) up to that time. However, a user cannot be assured of any given request I/O operation taking precedence over any previously issued Request I/O instructions because it cannot be determined (unless an error exists or the LUD is suspended) whether processing on that previous request has already been started. Whether or not processing on a request has started can be determined when an error condition is encountered for which the feedback record is marked terminating error. Then, after the user causes all feedback records to be dequeued from the response queue, remaining request I/O operations outstanding are not started, and subsequent Request I/O instructions are enqueued according to key sequence with the outstanding requests until a Request I/O (continue) instruction causes resumption of normal processing.

Dequeue of messages from the request I/O response queue can be done with any key compare operand supported by the Dequeue instruction. These operations are allowed because the request I/O operation sends messages to this queue in keyed sequence and time of arrival sequence within key just like a normal enqueue to a keyed queue. A Dequeue instruction with a less than (<) compare operand retrieves messages in key sequence, FIFO within key.

Request I/O and dequeue operations are asynchronous because there is no established relationship as to when a Dequeue instruction should be issued to receive the results of a previous Request I/O instruction. (The Dequeue instruction can be deferred as long as desired after the Request I/O is issued.) Conventional dequeue and branch, or dequeue and wait techniques must be applied as in any other queue relationship.

Special Considerations for the Request I/O Response Queue

The request I/O response queue must be created and defined as any other queue, except it must be defined with certain attributes. Refer to *Queue Management* in Chapter 5 for more information about how to create and define queues.

When a user creates a request I/O response queue, the following restrictions apply:

- The queue must be a keyed message queue (rather than LIFO or FIFO).
- The length of the key can be from 10 through 256 bytes.
- The messages must allow pointer and scalar data.
- The size of the message data area must be a minimum of 64 bytes.

When any of these restrictions are violated, the Request I/O instructions that reference this response queue signal an exception at the time the instruction is executed (before the request is processed).

The response queue can be created as either fixed in size or extendible. In either case the request I/O operation signals an exception when the queue message limit is violated (for fixed size queues) or when the machine storage limit is exceeded (for extendible queues).

When a message is dequeued from a request I/O response queue, the message contains the request ID information from the SSR for this request. All elements within the message prefix are as defined for other queues. The enqueue time stamp element represents the time when the Request I/O instruction was executed.

Format and Contents of the Feedback Record

The message associated with a Dequeue instruction is called a feedback record whenever the enqueued message is a result of a request I/O operation. The space that is specified as an operand on the Dequeue instruction has the following information inserted into it to form the feedback record.

- Space pointer—identifies the SSR (source/sink request) that was supplied by the issuer of the Request I/O instruction as its space operand. This SSR can optionally have new data inserted into it by the machine (based on the request I/O operation that was performed).
- Request ID—is the same as the one in the original SSR.
- Error summary field—indicates the final disposition of the request I/O operation.
- RD number—indicates the request descriptor within the Request I/O instruction that is appropriate for the ending status of that instruction. Normally, it is the last RD in the request and, in terminating errors, it is the RD on which the failure occurred.
- RIU segment count—indicates a further breakdown to the segment within the RIU (request information unit) associated with the above RD number.
- Device-dependent status field—indicates further status associated with the error summary field and is defined uniquely for each type of device supported on the system.

SOURCE/SINK EXCEPTIONS AND EVENTS

Some of the exceptions and events associated with source/sink instructions are applicable throughout the system. Other exceptions and events, which are system-wide, can have unique meanings when applied to source/sink objects; still others are uniquely defined for the source/sink objects. For a complete description of all exceptions and events for each instruction, refer to the *System/38 Functional Reference Manual*.

COMMUNICATIONS ERROR RECOVERY

Communications error recovery is available to the machine interface (MI) user of communications or local work stations to reestablish a connection and/or recontact a remote device or system after a line, station, device, or LUD failure has occurred. A Modify ND/CD/LUD (vary off) and Modify ND/CD/LUD (vary on) instruction sequence is not always required to reestablish a connection and/or recontact a remote device or system.

During a connection, errors may occur because of a line/station failure or a device/LUD failure. For these failures, the MI user is notified through events and/or an error request I/O feedback record. The failure events contain data that indicates the line, controller, device, or LUD on which the failure occurred and the cause of the failure. The error request I/O feedback record indicates the cause of the failure. When notification of a failure is given through a Request I/O Feedback Record, the MI user must issue the Modify LUD (reset) and Modify LUD (de-activate) instructions.

To reestablish the connection and contact the remote device or system after a line, station, or device/LUD failure, the MI user must issue an appropriate Modify ND (continue), Modify CD (continue), or Modify LUD (continue) instruction. The Modify ND (continue) instruction allows the line to be reused after a line failure. The Modify CD (continue) instruction allows the controller and its attached devices/LUDs to be reused after a line or station failure by reestablishing contact with the controller and its attached devices/LUDs.

The Modify LUD (continue) instruction allows the device/LUD to be reused after a device/LUD failure by reestablishing contact with the device/LUD.

To suspend the reconnection and reuse of the communications connection after a line, station, or device/LUD failure, the MI user must issue an appropriate Modify ND (cancel), Modify CD (cancel), or Modify LUD (cancel) instruction.

To reestablish the connection and contact the remote device or system after a Modify ND/CD/LUD (cancel) instruction has been issued, an appropriate Modify ND (continue), Modify CD (continue), or Modify LUD (continue) instruction must be issued. The Modify ND (cancel) instruction is used to suspend the reuse of a line after a line failure. The Modify CD (cancel) instruction is used to suspend the reuse of a controller and its attached devices/LUDs after a line or station failure. The Modify LUD (cancel) instruction is used to suspend the reuse of a device/LUD after a device/LUD failure.

SYSTEMS NETWORK ARCHITECTURE CONCEPTS FOR SYSTEM/38

A key function of SNA is the division of the communications system functions into a set of well-defined logical layers. The major functional layers defined by SNA are:

- Application layer
- Function management layer
- Transmission management layer

These functions are performed by either the machine (indicated by MI) or the controlling program (indicated by PGM).

Application Layer (PGM)

The application layer is only concerned with application functions. This layer performs the user's application processing in such a manner that the user need not be involved in the protocols or the procedures for controlling a communications line or routing data units through the network.

Function Management Layer (PGM)

The application layer employs a set of requests to invoke the services of the function management layer. The function management layer is concerned with the presentation of information between the application layer and the transmission management layer.

Transmission Management Layer (MI)

The transmission management layer is concerned with the routing and movement of data units between origins and destinations. The transmission management layer does not examine, use, or change the contents of the data units passed from the function management layer. This separation, where the routing of a data unit is independent of the contents of the data unit, means that a change in the method of transmission between units requires no change in the data unit itself. Therefore, the support provided by the function management layer can be used across a variety of physical connections. The paths through the network can be shared by many applications. The transmission management layer provides the control necessary to manage these shared resources.

SNA Transmission Management Layer

In SNA terminology, the transmission management layer is called the transmission subsystem.

Transmission Subsystem (MI)

The transmission subsystem provides information exchange between NAUs (network addressable units) and the three types of elements (data link control, path control, and transmission control) that make up the transmission subsystem.

Data link control elements manage the links between the units. Path control elements provide routing of data units over the paths between network addresses. Collectively, the path control and data link control elements make up the shared common network.

Transmission control coordinates the transmissions, including sequence numbering and rate control, when two network addressable units are connected.

Data Link Control (MI)

The data link control elements manage an individual data link. The data link control in each unit manages the data link attached to that unit. Data link control may function as either a primary or secondary station, or as a peer connection for X.25 (depending upon the physical configuration). The procedures and protocols used to transmit data depend on the type of data link being controlled.

Advanced Program-to-Program Communications

Advanced program-to-program communications (APPC) is an extension of the existing communications support. Instead of the more traditional host/work station relationship, APPC allows a System/38 to communicate not only with another System/38, but also with other systems that have compatible support through SNA/SDLC or SNA/X.25 on a peer (equal) basis. System/38 APPC uses the SNA-defined logical unit 6.2 (LU6.2), which allows either system to start program-to-program sessions and to initiate programs on the remote system.

In the APPC support, logical unit descriptions (LUDs) assume a new role. For example, an LUD does not represent a device such as a 5250. In APPC, an LUD represents a group of parallel paths to another processing unit; when more than one group of parallel paths are attached to a CD, they represent a set of parallel independent path groups to another processing unit.

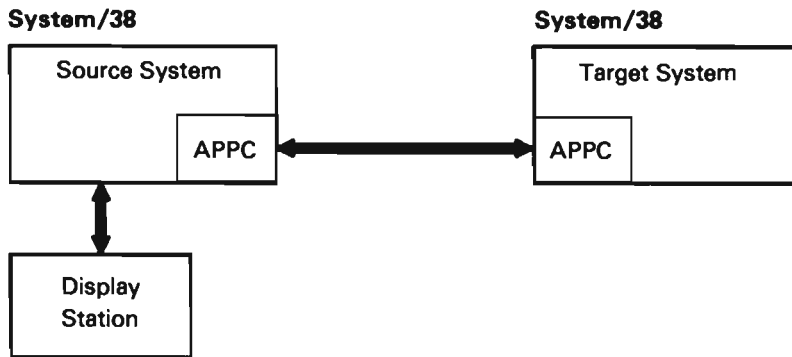
These paths are referred to as assignable sessions. This means that, in APPC, an LUD contains no device-unique data in it such as model number or screen size, but it does have unique SNA bind information. When a user program opens a file with an LUD defined for APPC use, a conversation on an available session is allocated to that file. Other sessions on the same LUD are still available to be used by other files under the same or other processes.

The LUD must be configured, through the configuration commands, to represent the conversations and sessions by specifying the maximum number of conversations and sessions to be run concurrently under that LUD. The LUD must also specify the number of sessions to bind automatically when the LUD is varied online and contact is made with the other system (PREBNDSSN = number of prebound sessions). The maximum number of sessions that this side can bind is also specified (MAXSRCSSN = maximum source sessions). If the maximum number of source sessions is greater than the number of prebound sessions, the remainder of the sessions are reserved for either dynamic initiation or initiation by the remote processing unit. An APPC logical unit description (LUD) can be attached to either PU2 controllers or peer controllers (APPC). When the APPC LUD is attached to a PU2 controller, only one session and one active conversation is allowed per device.

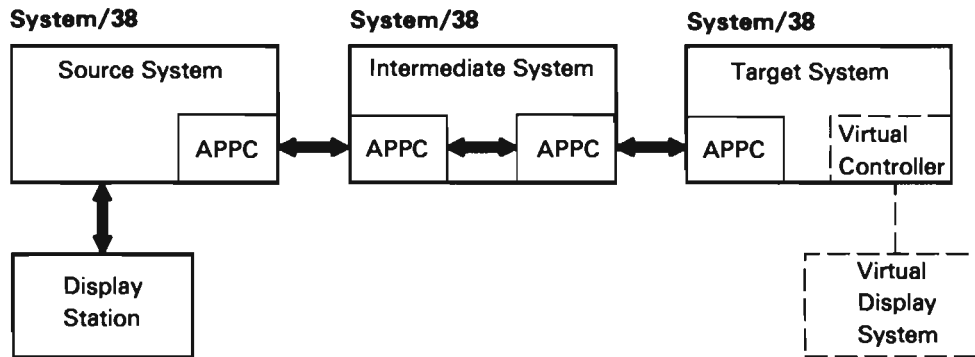
In APPC, a controller description (CD) describes the peer processing unit characteristics, SNA transmission parameters, and the peer processing unit configuration. The peer processing unit characteristics consist of its station address and its connection technique. The SNA parameters indicate the physical unit type, number of buffers, size of the buffers, and the transmission header type. The peer processing unit configuration consists of a chained list of logical units attached to the CD. The logical units can only be APPC peer LUDs.

Display Station Pass Through

Display station pass-through provides a convenient means for a user at a source system to sign on a target system. This System/38 function uses the APPC link, and is always initiated at a display station that is physically attached (either local or remote) to the source system. When the source and target systems are attached (via the APPC link), the display station at the source system appears to be physically attached to the target system.



There can also be one or more intermediate systems between the source and target systems. For example:

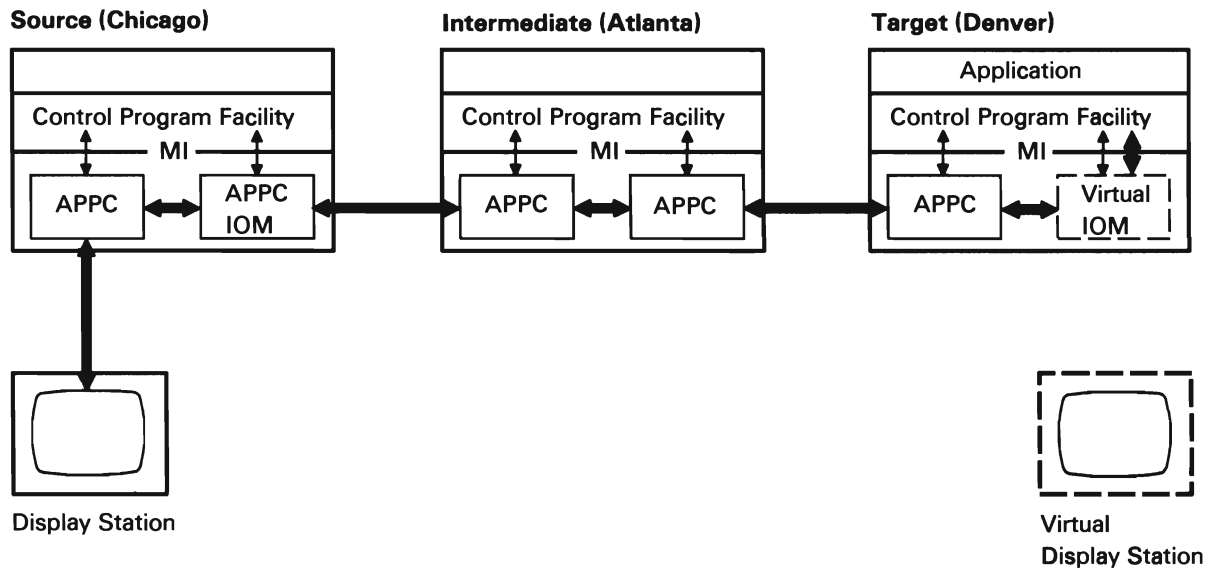


The source and target systems must each contain a logical unit description that describes a display station. However, the logical unit description at the source system describes a physical device, while the logical unit description at the target system describes a logical (virtual) device.

The following is an overview of a display station pass-through operation.

The pass-through link will connect a System/38 in Chicago, one in Atlanta, and one in Denver.

Before a display station pass-through operation can occur, all logical unit descriptions, controller descriptions, and network descriptions associated with the pass-through operation must be in a vary on state.



The Request Path Operation instruction is used for multiple purposes. Two of these purposes are for starting and for stopping a pass-through operation.

When the Request Path Operation instruction is issued at the source system (Chicago), communication is started between the display station I/O manager (IOM) and the advanced program-to-program IOM. A path now exists from the display station to the APPC I/O manager in the intermediate system (Atlanta).

The Request Path Operation instruction is now issued at the intermediate system, and communication is started between the two APPC I/O managers. A path now exists from the display station to the APPC I/O manager in the target system (Denver).

The Request Path Operation instruction is now issued at the target system, and communication is started between the APPC I/O manager and the virtual I/O manager. A path now exists from the display station to the virtual I/O manager in the target system. The display station appears to be locally attached to the target system because it is represented at the target system by a virtual logical unit description.

The application program in the target system can now communicate with the display station by accessing the virtual logical unit description.

The pass-through operation is normally terminated by the target system issuing another REQPO instruction. However, any source, intermediate, or target system can terminate the pass-through operation by issuing the REQPO instruction.

For additional information about display station pass-through, refer to the *Functional Reference Manual—Volume 2*, and the *Data Communications Programmer's Guide*.

SNA Supervisory Services Support (MI)

SNA supervisory services designate the set of services provided within the machine to control the shared resources of the communications network. The machine services control point (MSCP) is the portion of the machine that provides services and coordinates service requests between users of the network.

The supervisory services represent those services necessary to support the SNA implementation on this system and do not necessarily imply support of all SNA functions.

Machine Services Control Point

The MSCP provides services and coordinates the processing of supervisory service requests for users of the source/sink resources. Service requests to the MSCP are divided into two categories:

- Implicit requests to support the source/sink instructions
- Explicit Request I/O instruction to the MSCP

The first category is applicable for all source/sink components because the MSCP provides assistance in supporting all the Modify (LUD, CD, and ND) instructions. These MSCP functions are not directly visible to the user but are included in the description of each individual instruction.

The second category is applicable only to SNA devices supporting or initiating supervisory service requests. Supervisory services, which are used to control the shared communications network resources, involve all components: LUs (logical units), PUs (physical units), the MSCP, and the user. Figure 6-7 shows the logical flow of requests and responses, and the mechanism for conveying these messages between the various components.

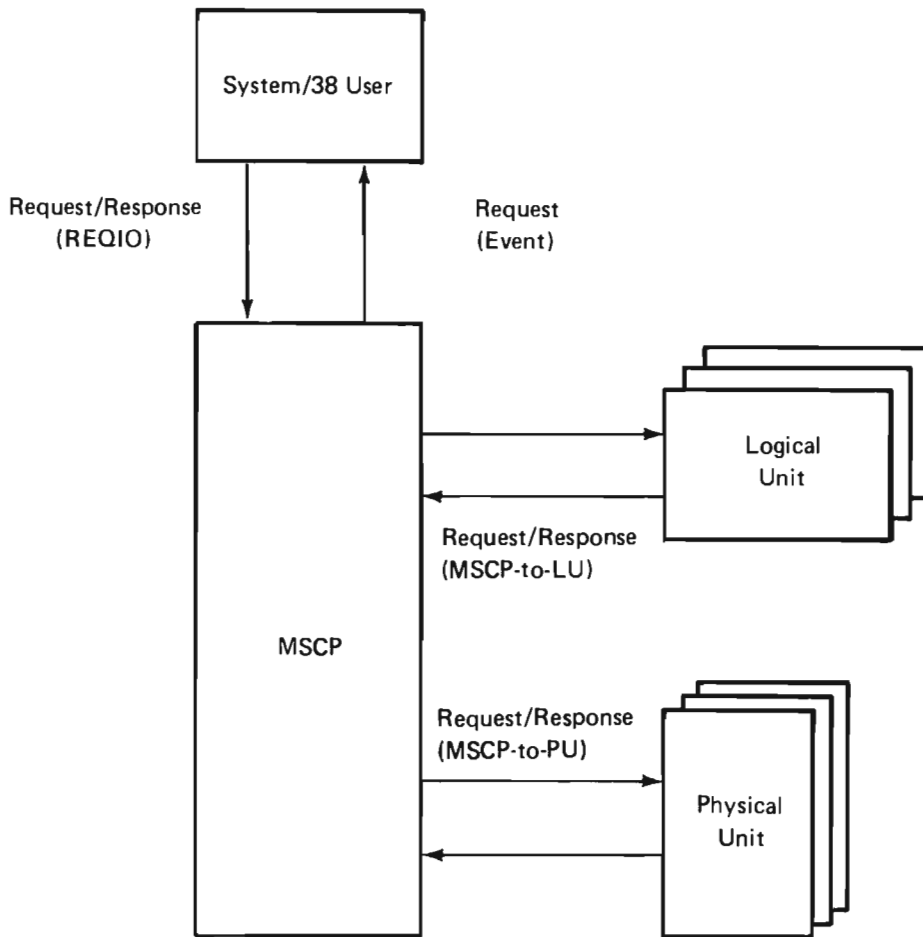


Figure 6-7. Supervisory Service Request Flow

The user directs requests or responses to the MSCP by using the Request I/O instruction with the function field set to indicate that it is an MSCP request. The format of the SSR (source/sink request) space and the SSD (source/sink data) space used with the MSCP REQIO instruction is the same format used for an REQIO instruction to SNA communications devices. The system pointer to a source/sink object in the SSR specifies an LUD or a CD to indicate the LU or PU with which the request is to be associated. The MSCP then uses an existing MSCP-to-LU or MSCP-to-PU session to route the message for action. An MSCP REQIO operation always results in a feedback record being enqueued to the request I/O response queue specified in the SSR. As an option, data can be placed in the byte space portion of the SSD.

The MSCP establishes contact with the remote controller by interacting with the System/38 SDLC support (primary or secondary line IOM) to establish a switched connection, if necessary, and to effect link-level identification with the remote station.

Knowledge of the current states of all network components is required in order to control the shared network resources and enforce network protocol. The MSCP maintains a record of the state of all logical units, physical units, and data links within its domain of control. In addition, MSCP knows the existence of sessions between users and logical units.

MSCP Role (Primary Station)

When the MSCP receives a request from an LU (logical unit) or a PU (physical unit) that requires interaction with a user, an event is signaled with supporting information made available. Supervisory service requests received by the MSCP from an LU as a result of operator action (for example, initiated via the System Request key) will be presented to the user. When the event is signaled to a user, the MSCP always sends a positive response to the initiator of the request.

MSCP to Physical Unit (Primary Station)

The MSCP is responsible for establishing and terminating sessions with the physical units it controls. If required, the MSCP sends the ACTPU (activate physical unit) session control message at the time the secondary station is initially contacted. The ACTPU establishes the MSCP-to-PU session and provides information about the rules that apply to the session. The ACTPU parameters are obtained from the appropriate entries in the CD. As long as the MSCP-to-PU session is active, the CD representing the physical unit is in the varied on state.

If the MSCP-to-PU session has been established, the MSCP sends the DACTPU (de-activate physical unit) control message when it is necessary to terminate the MSCP-to-PU session.

The MSCP-to-PU session is primarily used by the MSCP to control the orderly allocation and use of network components. However, an REQIO instruction to the MSCP with a system pointer resolved to a CD indicates that a message is to be sent on the MSCP-to-PU session.

MSCP to Logical Unit (Primary Station)

To establish an MSCP-to-LU session, the MSCP sends an ACTLU (activate logical unit) control message at the time the SNA device is physically in the varied on state. The ACTLU parameters are retrieved from the LUD and used to specify the rules to be enforced for users of this session.

While the MSCP-to-PU session is primarily used by the MSCP, the messages concerning the MSCP-to-LU session are usually directed between a user program and a logical unit operator. The MSCP is still responsible for coordinating requests and controlling messages concerning this session. Incoming requests are signaled as events, and outgoing requests are directed to the LU by providing the appropriate LUD system pointer (which is contained in the SSR of an REQIO instruction) to the MSCP.

The MSCP sends the DACTLU (de-activate logical unit) control message when it is necessary to terminate the MSCP-to-LU session.

MSCP Role (Secondary Station)

In a secondary role, the MSCP provides services for the logical unit (LU) and the physical unit (PU). The services consist of signaling events when control messages (such as ACTPU, ACTLU, DACTPU, and DACTLU) are received from the host system. When the event is signaled to a user, the MSCP sends either a positive or negative response to the sender of the control message.

Negative responses are sent for the following conditions:

- ACTPU control message.
 - CD (Control Description) is not in the varied on pending state.
 - SSCP ID does not match the ID contained within the CD.
 - The ND associated with the line is not in the ND candidate list of the CD.
- ACTLU control message.
 - The LUD (logical unit description) is not in a varied on or higher state.
 - The LUD (logical unit description) recovery/resource activation state is inoperative pending or normal cancel.

SSCP to Physical Unit (Secondary Station)

The host system (SSCP) is responsible for establishing and terminating sessions with the physical units (System/38) it controls. The ACTPU control message establishes the SSCP-to-PU session and also provides the rules that apply to the session. The MSCP provides the supervisory services that enforce the rules.

When an SSCP-to-PU session is established, the CD that represents the physical unit is moved to the varied on state. It remains in this state until it is explicitly varied off.

The SSCP-to-PU session terminates when the host system sends the DACTPU control message.

SSCP to Logical Unit (Secondary Station)

The host system (SSCP) is responsible for establishing and terminating an SSCP-to-LU session. The ACTLU control message establishes the session and also provides the rules that apply to the session. When the session is established, the LUD that represents the logical unit is moved to the varied on state where it remains until it is explicitly varied off. The SSCP-to-LU session becomes inactive within System/38 when the LUD is explicitly varied off. However, this session becomes active within the host system until a DACTLU control message is received at the secondary station. The SSCP-to-LU session resumes if the LUD is once again moved to the varied on state.

While the SSCP-to-PU session is primarily used by the MSCP, the messages (such a LOGON and LOGOFF) related to the SSCP-to-LU session are usually directed between a System/38 application program and a host system application program. However, the MSCP is still responsible for the messages concerning this session. Incoming requests are signaled as events; outgoing requests are directed to the LU by providing the appropriate LUD system pointer (which is contained in the SSR of an REQIO instruction) to the MSCP.

The SSCP-to-LU session terminates when the host system sends the DACTLU control message.

MSCP Role (Peer Station)

When System/38 is a peer station, the MSCP assumes one of three distinct roles, depending upon the role of System/38 at the SDLC level (primary or secondary) and the node type (2 or 4) of the remote controller. Type 2 nodes are the SNA PU.T2.1 defined facilities, and type 4 nodes are the SNA PU.T4 defined facilities.

System/38 Primary SDLC Station, Remote Controller Type 2 Node

The MSCP establishes contact with the remote controller by interacting with the primary line IOM to establish a switched connection, if necessary, and to obtain and verify the identity of the remote station via the SDLC XID sequence.

For this mode of peer operation, there are no MSCP-to-PU or MSCP-to-LU sessions. Instead, the user can activate any LUD attached to a CD when contact is made with the remote controller. Events are signaled to the user at this time for the CD and all LUDs attached to the CD that are in vary on pending or a higher state.

After an LUD is activated, the user (operator or application program) can place the LUD in the active session and establish LU-to-LU sessions.

System/38 Secondary SDLC Station, Remote Controller Type 2 Node

The MSCP establishes contact with the remote controller by interacting with the secondary line IOM to establish a switched connection, if necessary, and to obtain and verify the identity of the remote station via the SDLC XID sequence.

For this mode of peer operation, there are no MSCP-to-PU or MSCP-to-LU sessions. Instead, the user can activate any LUD attached to a CD when contact is made with the remote controller. Events are signaled to the user at this time for the CD and all LUDs attached to the CD that are in vary on pending or a higher state.

After an LUD is activated, the user (operator or application program) can place the LUD in the active session and establish LU-to-LU sessions.

System/38 Secondary SDLC Station, Remote Controller Type 4 Node

The MSCP establishes contact with the remote controller by interacting with the secondary line IOM to establish a switched connection, if necessary, and to obtain and verify the identity of the remote controller via the SSCP-ID in the ACTPU command sent by the remote SSCP.

The control messages ACTPU, ACTLU, DACTPU, and DACTLU are sent by the SSCP and are processed by System/38 in the same way as previously described under *MSCP Role (Secondary Station)*.

The connection becomes a peer connection only after the SSCP-to-LU session has been established. The LOGON and LOGOFF messages are not used. Instead, LU-to-LU sessions are established by the remote system (for example, CICS), and these sessions use the LU 6.2 protocol.

For more information about System/38 communications facilities, refer to the *Data Communications Programmer's Guide*.

BINARY SYNCHRONOUS COMMUNICATIONS CONCEPTS FOR SYSTEM/38

The System/38 BSC support contains some of the same organizational concepts as SNA. For example, both BSC and SNA have an application layer and a function management layer.

The basic layers for the BSC communications functions are:

- Application layer
- Function management layer
- I/O management layer
 - BSC
 - MTAM (MULTI-LEAVING telecommunications access method)

These functions are performed either by the machine (indicated by MI) or by the application program (indicated by PGM).

Application Layer (PGM)

The application layer is only concerned with application functions. This layer performs the user-application processing in such a manner that the user need not be involved in the protocols or the procedures for controlling a communications line or routing data units through the network.

Function Management Layer (PGM)

The application layer employs a set of requests to invoke the services of the function management layer. The function management layer is concerned with the presentation of information between the application layer and the I/O management layer. Some of the functions provided by the function management layer are:

- Prepares the file for I/O operations
- Performs record blocking or deblocking
- Performs record compression or decompression

I/O Management Layer (MI)

System/38 supports the BSC point-to-point, BSC tributary, BSC MULTI-LEAVING telecommunications access method protocols (modes), and 3270 emulation using multipoint tributary.

Each BSC protocol requires a unique IOM (I/O manager). However, even though they are unique, some functions are supported by all BSC IOMs. They are:

- Performs error recovery—including hardware errors
- Assists in establishing switched network connections
- Logs errors
- Keeps statistics about the communications link operation

Functions that are supported for each BSC protocol are:

- BSC point-to-point
 - Point-to-point switched and nonswitched lines
 - One LUD attached to one controller description
 - One controller description attached to one line description
 - EBCDIC to ASCII translation
 - Online test

- BSC tributary
 - BSC multipoint protocol as the tributary station on nonswitched lines
 - Multiple sessions—one session for each LUD
 - 32 LUDs attached to one controller description
 - One controller description attached to one line description
 - EBCDIC to ASCII translation
 - Online test (except for 3270 emulation)

- BSC MULTI-LEAVING telecommunications access method
 - MRJE binary synchronous communications with multiple reader, printer, and punch sessions supported
 - Point-to-point switched or nonswitched operations
 - Multiple LUDs attached to one controller description
 - EBCDIC character code

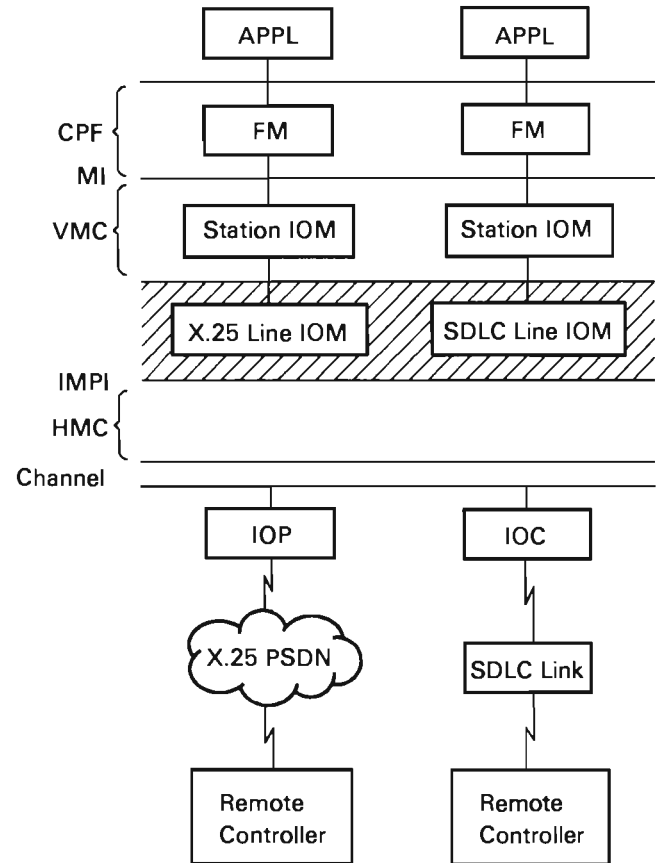
- BSC 3270 emulation
 - 3271 controller and 3277 display devices
 - Nonswitched lines
 - 32 LUDs attached to one controller description
 - One controller description attached to one line description
 - Program interface support
 - EBCDIC to ASCII translation

An IOM is active for each BSC line that is in the varied on state.

X.25 COMMUNICATIONS CONCEPTS FOR SYSTEM/38

The System/38 X.25 support replaces the DLC layer of SNA (normally SDLC) with the CCITT-defined X.25 packet protocol.

The MI user is aware of X.25 only through the addition of a new ND type and through parameter changes within the CD. All existing SNA types will run on X.25.



AAJ001-0

The key features of System/38 X.25 support are:

- Two line connections (ports) supported with line speeds up to 64 kilobits, each.
- Up to 32 virtual circuits per port allowing up to 32 simultaneous SNA link connections (sessions); any mix of PVC and SVC.
- Support data packet sizes from 64 to 1024 bytes.
- Support for SNA primary and secondary CDs on the same ND.

LOAD/DUMP CONSIDERATIONS

The load/dump (LD) function enables the user to save (back up) certain permanent objects by dumping these objects to LD media (such as tape, diskette, or dump space). In addition, if objects are dumped to a dump space, and the dump space is then dumped to an external media, the original objects can be loaded back directly from the resulting media.

When the specified operation is load or dump, the RD (request descriptor) field in the SSR identifies the object(s) to be loaded or dumped.

The objects that can be processed by the LD function are data spaces, journal spaces, data space indexes, programs, space objects, dump spaces, and independent indexes.

These objects can be dumped, loaded over an existing object (except programs), or loaded onto a system where they currently do not exist (are not created). An existing program object cannot be overlaid (loaded) on the system unless the program is loaded with the Create and Load command.

The interface to LD is through the Request I/O, Modify LUD, and Request Path Operation instructions. REQIO and MODLUD are used when the target is offline media (such as tape or diskette). REQPO is used when the target is a dump space. For simplicity in the following section, only the REQIO and MODLUD instructions are mentioned. The following chart lists the REQPO instruction to correspond to REQIO and MODLUD.

When the text uses:	The equivalent REQPO function is:
Request I/O (Normal)	REQPO I/O Request
Request I/O (Continue)	REQPO I/O Request (Continue)
MODLUD Activate	REQPO Initiate Path
MODLUD Reset	REQPO Reset Path
MODLUD Suspend	REQPO Suspend Path
MODLUD Quiesce	REQPO Quiesce Path
MODLUD De-activate	REQPO Terminate Path

LD Commands

The LD function uses unique commands (which are specified in the request descriptor) to process the objects. Any LD command will process any LD object. The LD commands are as follows:

- **Dump command**—The Dump command copies an object to the LD medium (magnetic tape or diskette).
- **Load command**—The Load command copies an object from the LD medium to overlay an object on the system. The object ID field (in the request descriptor) is compared to the object ID on the LD medium. When a match is found, that object is loaded to the address contained in the pointer field of the RD (request descriptor). A value in the compare length of the RD specifies the number of positions to be compared.
- **Create and Load command**—The Create and Load command creates an object in the system. The current file on the LD medium is searched for the object to be created by comparing the object ID field in the RD to the object ID on the LD medium. When a match is found, the system allocates space for the object and then loads the object into this space. Addressability to the object is placed in the context specified by the previous Set Context command only if the context bit in the object control field in the RD is equal to binary 0. The LD function provides addressability to the created object by unconditionally inserting a system pointer in the pointer field of the RD. Ownership of the object is assigned to the user profile specified by the previous Set User Profile command. If changes to the object were being journaled when it was dumped and if the journaling bit in the object control field is a binary 0, then the changes for the object are journaled through the journal port that was specified on the previous Set Journal command.
- **Set User Profile command**—The Set User Profile command must precede any Create and Load command within a normal REQIO instruction because it specifies a user profile for the object(s) created by the Create and Load command. The pointer field in the RD must contain a system pointer that has addressability to the desired user profile. After a Set User Profile command is processed, all subsequent Create and Load commands use that user profile until another Set User Profile command is processed.
- **Set Context command**—The Set Context command selects a specific context that can receive addressability to the object(s) created by the Create and Load command. Within a normal REQIO instruction, the Set Context command must precede the first Create and Load command that has a value of binary 0 in the context bit of the object control field. This requirement exists because the binary 0 value directs the LD function to put addressability to the created object in a context. The pointer field in the RD must contain the address of this context. After a Set Context command is processed, all subsequent Create and Load commands with the context bit in the object control field equal to binary 0 use that same context until another Set Context command is processed.
- **Read Object ID command**—The Read Object ID command retrieves only the data in the ID portion of an object on the LD medium and inserts the data into the ID field of the request descriptor. The retrieved data can then be used to compile a listing of the objects (ID only) on the file.
- **Set Journal Command**—The Set Journal command selects the journal port through which the changes to certain objects are to be journaled. The Set Journal command should be issued before any Create and Load commands of objects whose changes were being journaled at the time they were dumped. When a Set Journal command is encountered, all subsequent Create and Load commands involving objects whose changes are being journaled (in the same Request I/O instruction) use that journal port until another Set Journal command is encountered.
- **Set Journal Data command**—The Set Journal Data command contains journal data that LD uses to build the entry-specific portion of journal entries. The Set Journal Data command should be issued prior to any LD command that causes journal entries to be made.

- **Set Load/Dump Parameters command**—The Set Load/Dump Parameters command communicates information about the Request I/O instruction from the machine interface to the load/dump function.

When the machine interface user defines network boundaries within a Request I/O instruction, the user must issue this command with the networking bit equal to binary 1 before any Load, Dump, or Create and Load commands are issued in subsequent request descriptors.

All subsequent Load, Dump, and Create and Load commands must have their network boundary bits set in the object control field to indicate network boundaries and non-network objects. The request descriptor in the Set Load/Dump Parameters command is not counted in the 8000-request-descriptor limit.

If the system does not encounter a Set Load/Dump Parameters command before encountering a Dump, Load, or a Create and Load command, the network boundary bits in the object control field on the Load, Dump, and Create and Load request descriptors are ignored. In this case, the load/dump function defines the network boundaries.

When the machine interface user wants to load objects out of a dump space that was saved to media, the user must issue the Set Load/Dump Parameters command with the 'load out of dumped dump space' bit set. When this bit is on, the media must be positioned at the start of a dump space. The remaining request descriptors in the Request I/O describe the objects originally saved in the dump space. The compress bits in the remaining request descriptors specify whether compression was used when dumping the objects into the dump space, not whether compression was used when dumping the dump space. If compression was used when dumping the dump space, the 'load out of dumped dump space' option will result in an 'invalid descriptor' feedback record summary.

If the system encounters a Set Load/Dump Parameters command after encountering a Dump, Load, or a Create and Load command, an exception is signaled.

Session Types

For the LD function, there can be two session types (dump or load). Only the Dump, Set Load/Dump Parameters, and Set Journal Data commands are allowed for the dump session; all LD commands, except Dump, are allowed for the load session. When the Read Object ID command (load session) is issued, the only other command type allowed in the Request I/O instruction is the Set Load/Dump Parameters command.

Sequence of Operation

The following is an overview of the load/dump function:

1. A load/dump modify LUD (activate) session is issued to open the load/dump session.
2. A Request I/O instruction is issued that points to an SSR that contains commands and pointers to the objects to be loaded or dumped from the LD medium.
3. The load/dump function checks the Request I/O instruction for errors and processes all commands.
4. The load/dump function sends a feedback message to the request I/O response queue to indicate the completion of the function.
5. A modify LUD (de-activate) session is issued to close the load/dump session.

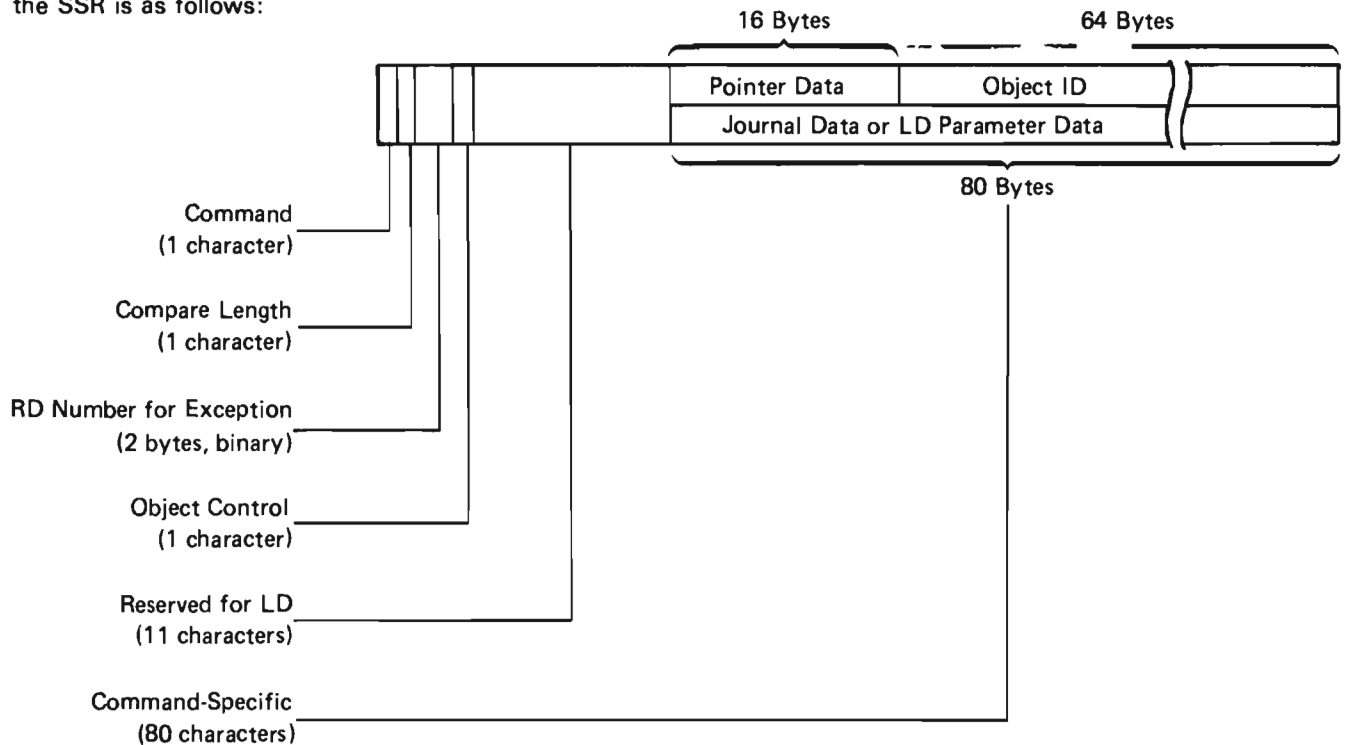
REQIO (Request I/O) Instruction

The user's interface to the load/dump function is the REQIO (Request I/O) instruction. Two types of REQIO instructions are used: the normal Request I/O and Request I/O (continue). The normal Request I/O instruction contains commands and the needed information to load objects to the system and dump objects from the system. The Request I/O (continue) instruction is used for recoverable error processing; it indicates that processing of the next normal Request I/O instruction should continue from the point where the error occurred. For continued operation, the user must reissue the SSR that contains the original unprocessed request descriptors associated with the Request I/O instruction that experienced the error.

If a nonrecoverable error is detected, a Modify LUD (reset) instruction must be issued. The LD function uses the standard format of the SSR (source/sink request) and an extended RD (request descriptor). An SSD (source/sink data) object is not used.

RD (Request Descriptor)

The format of the load/dump RD (request descriptor) in the SSR is as follows:



The length of each RD must be 96 bytes and located on a 16-byte boundary.

Each RD must contain the necessary information to process at least one object. For load/dump, the combined number of load, dump, and create and load RDs within the SSR is limited to 8000.

Request descriptors are always processed by the load/dump function in the order they appear in the source/sink request; that is, the first request descriptor is processed first, and the last request descriptor is processed last.

Command Field

The user must specify one of the following commands for each RD:

- Load
- Create and Load
- Dump
- Set User Profile
- Set Context
- Read Object ID
- Set Journal
- Set Journal Data
- Set Load/Dump Parameters

Compare Length Field

If the command is Load or Create and Load, the compare length field specifies how many bytes of the object ID on the LD medium are to be compared with the object ID field. The compare length can be any value from 0 through 64. A length of 0 indicates that the next object on the LD medium should be loaded. A length of 64 indicates that an exact match of the object ID is required before the load can occur.

If the command is Set Journal Data, the compare length field specifies the length of the journal data.

Note: Because load dump media is read serially, the user should exercise care when a compare length of less than 64 bytes is specified. This is because the device starts searching for a match on the ID field from where the device was last positioned. It is possible for the system to load the wrong object if the user does not know the exact data in the object IDs and the sequence of the objects on the device.

RD Number for Exception Field

If an exception error code is generated by an REQIO instruction, this field can contain the RD number that caused the exception. This field is used only in the first RD of the SSR.

Object Control Field

The contents of this field are meaningful only when the associated command is Dump, Load, Create and Load, or Read Object ID. For all other commands, this field is reserved and must be set to 0.

The context bit in this field determines whether addressability for the object being created will be placed into the context specified by the last Set Context command.

The journal bit in this field determines whether the LD function should continue the journal functions after the object is loaded. The journal functions are continued only if changes to the object were being journaled when the object was dumped. The changes are journaled through the journal port specified in the previous Set Journal command.

The compress bit in this field determines whether the LD function compresses or decompresses the data. If the compress bit is on, LD compresses the data on a Dump command and decompresses the data on a Load, Create and Load, or Read Object ID command.

The network boundary bits in this field specify the relationship of an object to an LD network as follows:

- The object is not part of an LD network.
- The object defines the beginning boundary of an LD network.
- The object is a member of an LD network and is contained within the beginning and ending boundaries.
- The object defines the ending boundary of an LD network.

The load associated space bit in this field determines whether the associated space of this object will be loaded. If this bit is set on (that is, the associated space bit is not loaded), it is valid only for Load commands and will result in an exception if set on for a Dump, Create and Load, or Read Object ID command.

Command-Specific Field

The command-specific field has a length of 80 bytes and is used for multiple purposes. For some LD commands, the 16 leftmost bytes provide addressability to some of the objects associated with the LD function. For other LD commands, these 16 bytes contain data associated with a specific command.

The remaining 64 bytes of the command-specific field are used either for an object ID or for a continuation of the data associated with a specific command.

The object ID consists of an object name (30 characters), object type (1 character), object subtype (1 character), and an ID extension (32 characters).

The data contained in the command-specific field for each of the following LD commands is:

Dump Command: A system pointer to the object that is to be dumped and an object ID. The object name, type, and subtype must be supplied. The ID extension is optional; however, the entire object ID (64 bytes) is dumped with the object.

Load Command: A system pointer to the object that is to be overlaid by the object from the LD medium and an object ID. The object ID is used during the search of the LD medium for the correct object(s). The search operation compares the object ID on the LD medium to the object ID in the request descriptor until an equal condition occurs. The number of characters compared in the search operation is determined by the value in the compare length field of the request descriptor.

Create and Load Command: Any value (pointer or data) when the request descriptor is built and an object ID. The LD function inserts a system pointer after the object has been created and loaded. The pointer contains addressability to the created object; no authority is placed in this pointer.

The object ID is used during the search of the LD medium for the correct object(s). The search operation compares the object ID on the LD medium to the object ID in the request descriptor until an equal condition occurs. The number of characters compared in the search operation is determined by the value in the compare length field of the request descriptor.

Set User Profile Command: A system pointer to the user profile that is given ownership of the object(s) on all subsequent Create and Load commands within the same normal Request I/O instruction. The remaining 64 bytes are not used.

Set Context Command: A system pointer to the context where addressability can be inserted for the objects created by the Create and Load command. Addressability is inserted under control of the context bit in the object control field. The remaining 64 bytes are not used.

Read Object ID Command: A 64-byte object ID that is read from the LD medium and inserted into the command-specific field. The leftmost 16 bytes of the field are not used.

Set Journal Command: A system pointer to the journal port through which the changes to selected objects will be journaled. The remaining 64 bytes are not used.

Set Journal Data Command: Specific data associated with this command when the system is making journal entries.

Set Load/Dump Parameters Command: Specific data associated with this command. The networking bit in the leftmost byte determines whether LD network boundaries are explicitly defined by the user or implicitly defined by the system. The remaining 79 bytes are not used.

The following diagram is a summary of the data within the command-specific field.

Command	Command-Specific Field (80 bytes)	
Load	System Pointer (16 Bytes)	Object ID (64 bytes)
Create and Load	System Pointer (16 Bytes)	Object ID (64 bytes)
Dump	System Pointer (16 Bytes)	Object ID (64 bytes)
Set User Profile	System Pointer (16 Bytes)	Not used
Set Context	System Pointer (16 Bytes)	Not used
Read Object ID	Not Used	Object ID from LD Medium (64 bytes)
Set Journal	System Pointer (16 Bytes)	Not Used
Set Journal Data	Used by LD while making journal entries	
Set Load/Dump Parameters	Load/Dump Parameters	

Modify LUD Sessions for LD

The LD function conforms to the normal operation for the various modify LUD sessions except when the session is changed from load to dump or from dump to load. To change the sessions from load to dump or dump to load the user must:

1. Issue an MODLUD (de-activate) instruction.
2. Change the operation mode byte in the LUD.
3. Issue an MODLUD (activate) instruction.

The MODLUD (reset) session (which may be required after an error condition) causes the LD function to immediately stop processing the current normal REQIO instruction and send a feedback record for the associated normal REQIO instruction. Included in the feedback record is the proper error code and an indicator stating how many RDs have already been processed. LD then flushes the unprocessed REQIO instructions by sending feedback records for each REQIO instruction with the proper error code. After the LD queue has been flushed, the MODLUD (reset) operation is completed. The LD function will do a cleanup, if necessary, on the RD it was processing when the MODLUD (reset) request was received. The MODLUD (reset) session state can stop the LD medium at an abnormal position; therefore, it is the user's responsibility to correctly position the LD medium after an MODLUD (reset) instruction is processed.

The MODLUD (suspend) session is to interrupt the LD function so that processing can be halted. The modify LUD (suspend) session causes the load/dump function to stop processing the current Request I/O (normal) instruction on a network boundary. A feedback record is sent for the associated Request I/O (normal) instruction after the suspend session has occurred.

LD Error Processing

In LD processing, there are four types of errors: exceptions, severe errors (nonrecoverable), recoverable errors (such as end of volume and end of file), and other errors. How these errors are processed depends on the type of error encountered.

Exceptions

The exception errors are detected by the Request I/O instruction. They result in an exception being generated and sent to the user's program. At this time, I/O operations have not started. To recover from an exception, correct the error and retry the REQIO (Request I/O) instruction.

Severe Errors

Severe errors are nonrecoverable errors and they seldom occur until after an I/O operation has started. A Modify LUD (reset) or Modify LUD (de-activate) instruction must be issued if this type of error occurs. The normal Request I/O and Request I/O (continue) instructions will not be processed.

Recoverable Errors

When a recoverable error occurs (for example, end of volume or end of file), the LD function returns the feedback record with the status of the error. The load/dump function does not process any other normal Request I/O instructions until a Modify LUD (reset) or a Request I/O (continue) instruction is issued.

If the error can be corrected (for example, by positioning the LD medium at the beginning of the file), the same normal Request I/O instruction that encountered the error must be returned unmodified and ahead of (lower value in the request priority field of the SSR header) all other normal Request I/O instructions that have been previously issued. A Request I/O (continue) instruction must then be issued.

An MODLUD (reset) instruction must be issued when the error cannot be corrected.

Use the Request I/O (continue) instruction to cause load/dump function to finish processing the next normal Request I/O instruction from the point where the recoverable error occurred.

Other Errors

Other types of errors only indicate status. They do not terminate the LD function or take it into or out of error mode.

Processing an MODLUD (Reset) Instruction

During the processing of an MODLUD (reset) instruction after a severe error or a recoverable error, the LD function will execute a cleanup procedure on the object that was being processed when the error occurred, if necessary. The LD function then returns a feedback record for each pending REQIO instruction (unprocessed) and completes processing the MODLUD (reset) instruction. The LD function is then ready to process additional Request I/O instructions. (The user must reposition the LD medium at the correct starting location after an MODLUD (reset) instruction is processed.)

Cleanup Procedure

The cleanup procedure (if required) is performed only for the Load and the Create and Load commands.

Load Command: If the object was partially loaded at the time the error occurred, the object is flagged and logged as damaged because the data portion of the object is left in an unknown state.

Create and Load Command: If space for an object was already allocated when the error occurred, that space is destroyed.

Feedback Record

For every REQIO received, the LD function responds with a standard feedback record that contains the status of that REQIO. The feedback record is made visible to the user when a Dequeue instruction is processed.

Load/Dump Authority

The normal Request I/O instruction determines whether or not the proper authority is available for each load/dump command. If the proper authority is not available, an exception is generated.

Data Base and Load/Dump Networks

A data base network consists of one or more data space indexes and all the data spaces associated with each data space index. When a network is dumped, the LD function saves the information that links a network together.

A load/dump network consists of request descriptors that are grouped together. A load/dump network may contain one or more data base networks.

A load/dump network can be implicitly or explicitly defined. Load/dump networks are implicitly defined when the networking bit in a Set Load/Dump Parameters command is off (0) or when a Set Load/Dump Parameters command is not encountered before the first Dump, Load, or Create and Load command is encountered. In this case, the LD function compares the position of each object in the RD (request descriptor) list with that of other objects in the RD list. A network is started with the first data space in the RD list following a previous network. A network is ended with the RD that is immediately before the next non-data-base object or the next data space that follows a data space index. The networking bits in the object control field are set to reflect the network boundaries implicitly defined by the LD function.

The following restrictions apply when LD networks are implicitly defined.

- All data spaces must precede all associated data space indexes in a network.
- Intertwined networks can be dumped, loaded, or created and loaded as long as there are no non-data-base objects between them.

The user can explicitly define those groups of objects that are to be treated as a load/dump network. The networking bit (set load/dump parameters RD) and the network boundary bits (object control field) are used for this purpose. The value and meaning of these bits are as follows:

- Networking bit

Binary Value	Meaning
0	Network boundaries are not defined by the user.
1	Network boundaries are defined by the user.

- Network boundary bits

Binary Value	Meaning
00	The object is not part of an LD boundary.
01	The object defines the beginning boundary of an LD network.
11	The object is contained within an LD network.
10	The object defines the ending boundary of an LD network.

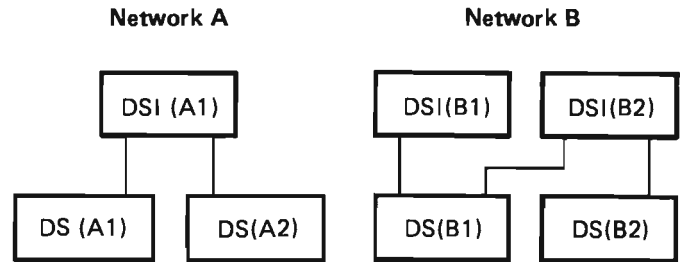
The following restrictions apply when LD networks are explicitly defined.

- Only one set load/dump parameters request descriptor is allowed for each Request I/O instruction.
- The set load/dump parameters request descriptor must precede all dump, load, and create and load request descriptors.
- The object control field must have a hex 00 value for all commands other than Dump, Load, or Create and Load.

- A non-data-base object can be defined as the beginning boundary of a load/dump network, as an object within a load/dump network, as independent of a network, or as the ending boundary of a load/dump network.
- In an LD network, all data space indexes must follow their associated data spaces. Data space indexes may precede data spaces with which they are not associated in the LD network.
- A load/dump network cannot contain duplicate objects.

The following example shows how the network boundary bits are used to define LD networks.

- Objects to be processed:
 - Space object
 - Network A
 - Data space (non-network)
 - Program
 - Network B



Network Boundary Bits (binary value)

Processed Objects

Network A	{	00	Space object
		01	DS (A1)
		11	DS (A2)
		10	DSI (A1)
Network B	{	00	Data space (non-network)
		00	Program
		01	DS (B1)
		11	DSI (B1)
		11	DS (B2)
	10	DSI (B2)	

The following restrictions apply when LD networks are implicitly or explicitly defined.

- Other (non-network) objects can be processed in the Request I/O instruction.
- LD networks are supported by all LD commands.
- A data space index cannot be dumped or loaded by itself; all of its associated data spaces must be dumped or loaded along with it.
- When a data space index within a network is loaded, the links to the data space(s) within the same network are connected.
- Any data space that is being loaded and all data space indexes over that data space must be loaded in the same Request I/O instruction; otherwise, the data space index is invalidated. An event is signaled whenever a data space index is invalidated.
- An active cursor cannot be over a data space or data space index on a Load command.
- When a data space is loaded, no data space indexes over that data space can be in use, damaged, or destroyed.
- When a data space index is loaded, the data space key specification in the data space index to be overlaid must match the data space key specification in the data space index to be loaded.
- A data space index is not loaded if any of the data spaces under the data space index are created and loaded.
- When a data space index is loaded, the same data spaces must be under both the data space index to be loaded and the data space index to be overlaid. These data spaces must also be in the same internal order in the data space index.
- A load/dump network cannot contain duplicate objects.

Load/Dump Performance

To achieve maximum performance from the LD function, the objects should be loaded in the same order as they were dumped. For example, if objects A, B, and C are dumped in alphabetic order, then objects A, B, and C should be loaded in the same order.

A group of objects defined as a network can be processed faster than if the objects are individually defined as independent.

Load/Dump Journal Entries

The LD function makes entries in a journal space about its operations. The following information discusses which entries are made for each LD command.

Dump Command

The object dumped entry is made in a journal space after an object has been successfully dumped. If the entry is not made, an event is signaled. The dump operation continues until it is successfully completed. The LD function continues processing the Request I/O instruction.

Load Command

The object loaded entry is made in a journal space after the load operation has started but before the object becomes available.

If the load operation fails after the object loaded entry is made, the entry remains in the journal space. The object may be marked as damaged.

If the object loaded entry is not made, an event is signaled and the load operation is terminated with an unrecoverable error.

Create and Load Command

The start journaling object entry is made in a journal space when LD determines that changes to the object can be journaled. The journal space in which the entry is made was attached to the journal port specified in the previous Set Journal command request descriptor.

The object loaded entry is made immediately after journaling is started for the loaded object.

If a failure occurs after journaling for the loaded object has started, LD attempts to make an object destroyed entry in a journal space. This action, along with the destruction of the partially loaded object, occurs during the execution of a Modify LUD (reset) instruction.

If neither entry, start journaling object nor object loaded, is made in a journal space, an event is signaled and the create and load operation is terminated with an unrecoverable error. If the event indicates that some type of recovery action can be initiated, the action should occur before the Modify LUD (reset) instruction is issued. If this sequence is followed, the possibility is increased that LD will be able to successfully make an object destroyed entry in a journal space.

Dumping and Loading Journal Spaces

Attached or unattached journal spaces can be dumped but only unattached journal spaces can be overlaid during a load operation. All of the entries in an unattached journal space are dumped. However, those entries that are added to an attached journal space after a dump operation has started are not dumped.

Dumping Unattached Journal Spaces

Journal spaces can be dumped either before they are attached to or after they are detached from a journal port. All of the entries in a journal space are dumped if the journal space is detached prior to the dump operation.

Loading Unattached Journal Spaces

Any journal space that was empty at the time it was dumped will still be empty after the load or create and load operation.

A journal space is in a detached state after a load or create and load operation regardless of whether it was detached or attached at the time it was dumped. The journal space being overlaid must either be empty or match the journal space on the LD medium. They match if all of the following are true:

- The starting sequence numbers are equal.
- The ending sequence number is less than or equal to that of the journal space on the LD medium. This check is done to avoid overlaying a newer version of a journal space with a back-level copy.
- The prefix lengths are equal.

Dumping Attached Journal Spaces

Journal spaces can be dumped while they are attached to a journal port. The number of entries to be dumped is determined at the time LD starts processing the attached journal space. The journal port with which the journal space is associated becomes unavailable while LD is determining the number of entries to be dumped. Entries that are added to the journal space after the dump operation has started are not dumped. The journal space cannot be destroyed or detached while it is being dumped.

Dumping attached journal spaces can result in some unpredictable situations. These situations occur because the attached journal space can change between the time LD determines how much is to be dumped and the time at which the journal space is written onto the LD medium. For example, the journal space can increase in size as more entries are added, become damaged, or become partially damaged.

New entries can be added to the attached journal space before the dump operation is completed. LD dumps only the entries that were present when it started processing the attached journal space. The new entries are not dumped.

The attached journal space could become damaged or partially damaged while it is being dumped. A damaged object cannot be accessed; a partially damaged object can be accessed by some operations but not all operations. If LD encounters the condition that caused the damage, the dump operation will fail. If LD does not encounter the condition that caused the damage, the dump operation will be successfully completed. In some cases the journal space on the medium is marked as damaged or partially damaged. The load operation of such a journal space will be successfully completed. A partially damaged journal space remains partially damaged after the load operation is completed. Because the journal space will be detached after the load, the only operation not allowed because of the partial damage is a dump. A journal space that is marked as damaged on the medium will be repaired after the load and will no longer be damaged.

SOURCE/SINK OBJECT RECOVERY

IPL Cleanup

Source/sink management performs the following cleanup operations on a source/sink object when the object is first used after an IPL.

ND, CD, and LUD Objects

The diagnostic active indicator and partial damage indicator are set off for ND, CD, and LUD objects.

ND Object

The ND active count is set to binary zero, the object state field is set to varied off, and the object recovery/resource activation field is set to a normal continue. The switched backward connection pointer is set to binary zero.

CD Object

The CD session count and the CD active count entries in the state change/status field are set to binary zero. The object state field is set to either varied off state or powered off state depending on the value in the power control field, and the object recovery/resource activation field is set to normal continue.

LUD Object

The object state field is set to either powered off state or powered on/varied off state depending on the value in the power control field, and the object recovery/resource activation field is set to normal continue. The load/dump pending indicator is set to binary zero.

Damaged Objects

If damage is detected and it is related to a source/sink object, the object is marked damaged, and an exception and an event are signaled. Normal recovery from damaged objects is to destroy the object. Destroy CD, ND, and LUD instructions are tolerant of damage; that is, the Destroy instruction is executed regardless of the damage to the object.

In some cases, the entire object network must be destroyed before the objects can be recreated and properly chained. If damage occurs while the objects are in a varied on state, an IPL may have to be performed (this sets the objects to the varied off state) before some of the undamaged objects in the network can be destroyed.

Damage Set by Source/Sink Instructions

Materialize instructions set object damage whenever an invalid forward pointer is detected. These instructions set partial object damage whenever invalid switched forward/backward pointers are detected.

The Destroy and Create instructions do not set object damage.

The Modify instructions set object damage if invalid forward pointers are detected.

The Request I/O instruction does not set object damage.

Partial Damage

Partial damage is set in an LUD, a CD, or an ND when an IOM (I/O manager) malfunction is detected, an invalid switched forward/backward pointer is detected, or a Modify instruction time-out occurs.

When an IOM malfunction is detected, the object associated with the IOM is marked partially damaged, and a partial damage event is signaled. If the object is a CD, the LUDs attached to the CD may also be marked partially damaged, and a partial damage event is signaled for each attached LUD.

When invalid switched forward/backward pointers are detected by the Materialize or Modify instruction, partial damage is set, and an exception is signaled. If the invalid pointer is detected by a modify operation that is tolerant to partial damage, then the partial damage is set, and the event is signaled; however, the exception is not signaled, and the modify operation is completed.

Some of the operations performed by a Modify instruction are timed. This ensures that the operation is properly completed. The user can optionally supply the time-out value in the modify template; however, if this value is not user-supplied, a default value is assumed. The minimum, maximum, and default values are contained in the *System/38 Functional Reference Manual, Volume 1*. If a time-out occurs during a modify operation, a partial damage exception and event are signaled. The object is marked as partially damaged.

Partial Damage Recovery

Partial object damage is reset by returning the object to the varied off state. The following modify state change functions are tolerant of partial damage:

- Modify LUD
 - Reset
 - De-activate (when issued after a Reset command in active state, from quiesced state, from suspended state, or from reset state)
 - Vary off
- Modify CD
 - Abandon connection
 - Vary off
- Modify ND
 - Abandon call
 - Vary off

These state change functions, in addition to being tolerant of partial damage, do not signal partial damage exception. In some cases, a partial damage event may be signaled, but the Modify instruction is completed without a partial damage exception being signaled.

An attempt to use the state change functions when partial damage is set (other than those previously listed) results in a partial damage exception being signaled. All state change rules must be followed when recovering from partial damage. For example, before a CD can be varied off, all LUDs under that CD must be varied off. State change violations result in the same exceptions regardless of whether or not partial damage is set.

All source/sink objects associated with an attempt to recover from partial damage should first be set in the varied off state. Then the recovery attempt can be made. For example, during the recovery operation from a partially damaged LUD, another partial damaged exception may be encountered when the LUD is varied on even though it was first varied off. In this case, the CD to which this LUD is attached must also be varied off to recover from the partial damage condition.

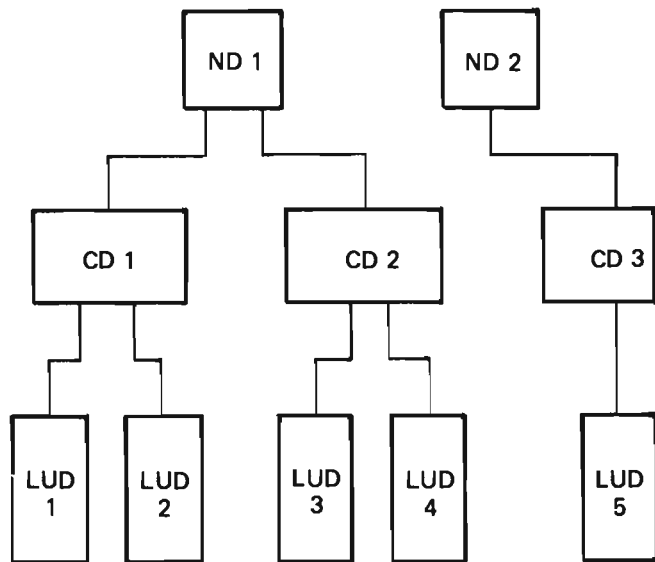
SOURCE/SINK EXAMPLES

Shared Usage of Source/Sink Objects

The use of source/sink objects and instructions can involve many different users and application programs within an installation. Generally, a user has three distinct responsibilities.


The first responsibility is the definition and management of the I/O configuration of the system. This responsibility is shared between the user and IBM support personnel. The I/O configuration is defined and managed through description objects (ND, CD, and LUD). To limit the definition and management functions to selected personnel, the instructions to create description objects are privileged. After the creation of description objects, the person authorized to issue the privileged Create instruction can allow other personnel to manipulate the created description object by granting the necessary functional authority to those persons.

Figure 6-8 shows the relationship that is established between LUDs, CDs, and NDs. These objects are normally created in order from the top down. The ND represents the characteristics of a specific I/O port on the system including local channel attachment or communications lines, modem characteristics, and line protocols used. The CD represents the characteristics of the controller (either a device control unit or a communications controller). The LUD represents the characteristics of the device itself. The vertical connecting lines in the figure represent forward and backward description object pointers contained within each object.




ND—Network Description
CD—Controller Description
LUD—Logical Unit Description

Figure 6-8. Source/Sink Structure




The second responsibility related to source/sink facilities is system administration and system operation. This responsibility relates to activating the system for scheduled usage. For example, activating the system includes modifying the description objects to perform such functions as power up devices, bring devices logically online, and IPL devices or controllers.

The third responsibility is to produce the set of application programs that conduct either I/O operations or load/dump functions with logical units. These application programs are not concerned with the configuration or activation details of the network. These programs modify the logical units to define the device operational characteristics appropriate for this session (for example: batch or interactive session, record sizes to be used, forms control, and chain type to be used by a printer). These programs issue all of the Request I/O instructions to the devices (logical units) for performing and controlling either the I/O operations or the load/dump functions.



Each user has the additional responsibility to properly terminate the appropriate elements. For example, programs must terminate sessions when operations are complete; system operations must also modify description objects for orderly shutdown of systems.



Creating, modifying, and destroying description objects when devices are added to, modified, or removed from the system are related to the first responsibility because this activity is considered to be managing the I/O configuration.

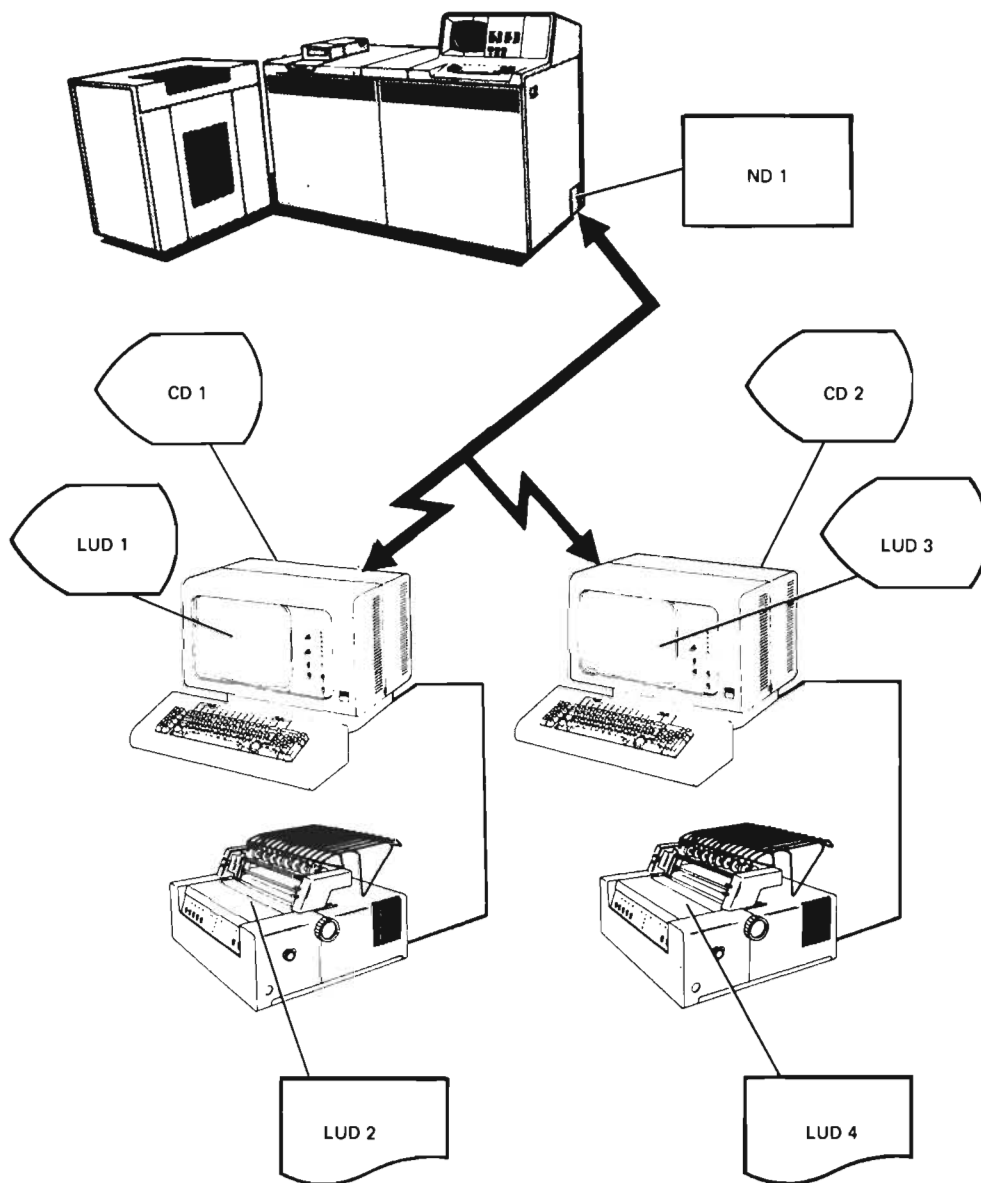
Configuration Changes

Previous information in this chapter explained the creation and destruction of source/sink objects and the hierarchical considerations for building or altering an I/O configuration. That information defined a preferred sequence for the creation and destruction of source/sink objects that minimizes the user's involvement in supplying configuration linkage through forward object system pointers and backward object system pointer lists. Previous information also explained how to destroy and create objects out of sequence at the expense of additional input data requirements and additional exception handling reconfiguring an I/O configuration. The following examples show the alternative methods available for creating networks and also show when it is advantageous to operate out of the normal sequence. These examples deal only with forward and backward pointer chaining and do not consider other elements within the objects.

Adding a Data Link

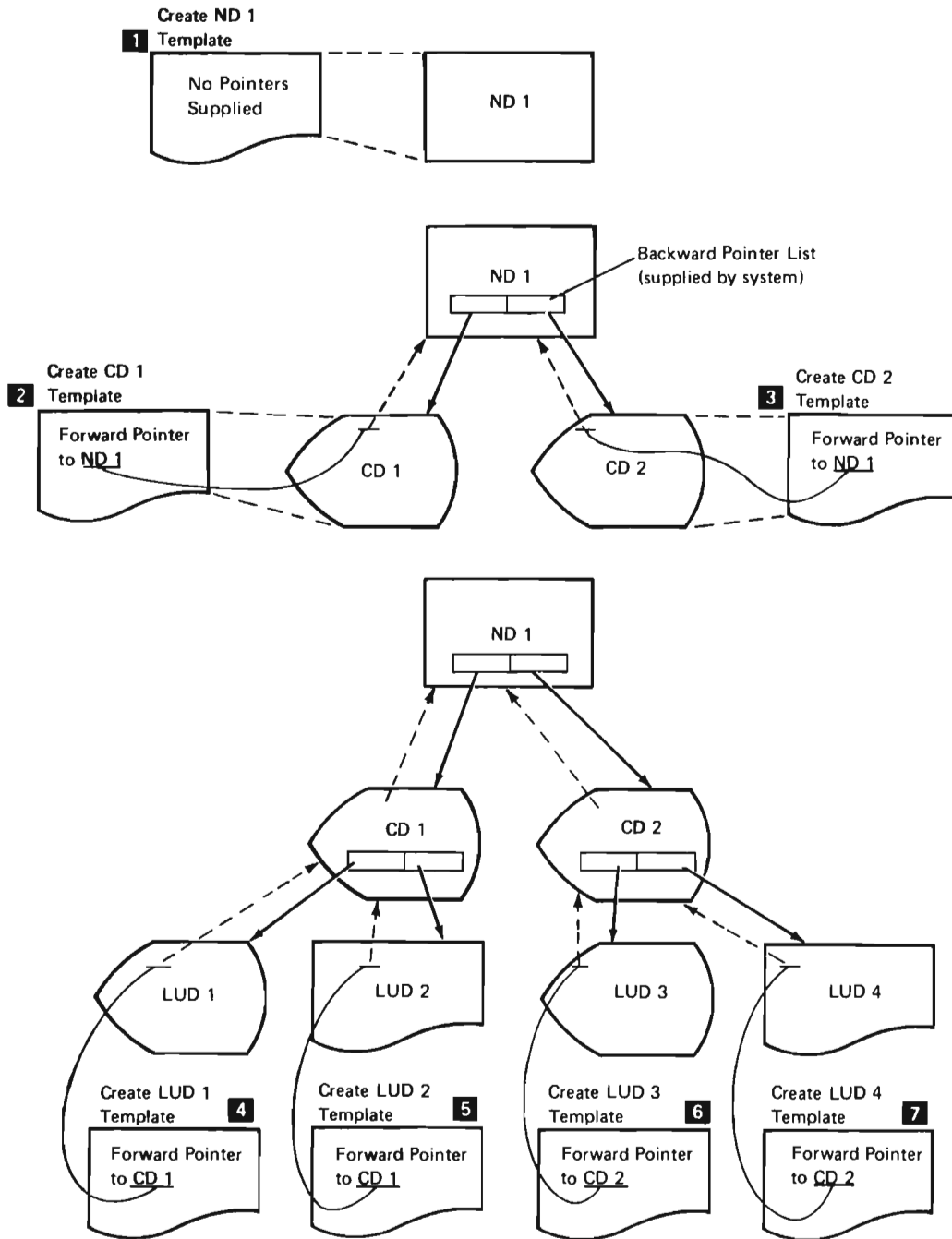
This example assumes that a new data link with two work stations (each having a display and a printer) is to be added to the system. This new link is represented by:

- ND 1 I/O port on system
- CD 1 Station 1
- LUD 1 Display on station 1
- LUD 2 Printer on station 1
- CD 2 Station 2
- LUD 3 Display on station 2
- LUD 4 Printer on station 2



Creating source/sink objects in sequence:

Operation		PTRs Supplied by User	Resulting Action by System
1	CRTND (1)	None	ND 1 created
2	CRTCD (1)	PTR to ND 1	CD 1 created, CD 1 PTR inserted into ND 1 backward list
3	CRTCD (2)	PTR to ND 1	CD 2 created, CD 2 PTR inserted into ND 1 backward list
4	CRTLUD (1)	PTR to CD 1	LUD 1 created, LUD 1 PTR inserted into CD 1 backward list
5	CRTLUD (2)	PTR to CD 1	LUD 2 created, LUD 2 PTR inserted into CD 1 backward list
6	CRTLUD (3)	PTR to CD 2	LUD 3 created, LUD 3 PTR inserted into CD 2 backward list
7	CRTLUD (4)	PTR to CD 2	LUD 4 created, LUD 4 PTR inserted into CD 2 backward list



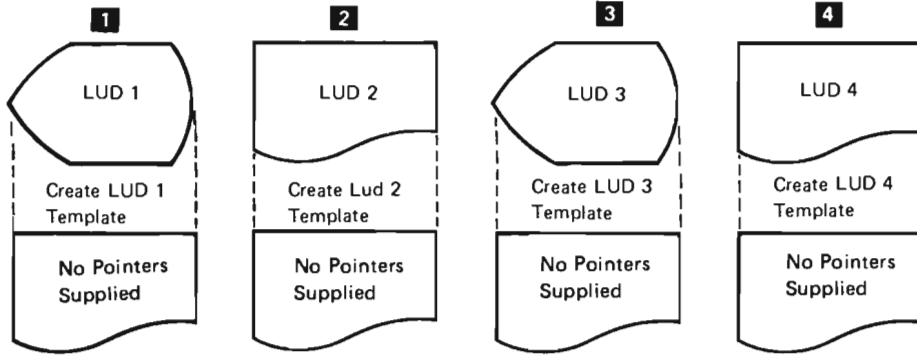
Legend:

- > Pointer supplied by user
- > Pointer generated by system

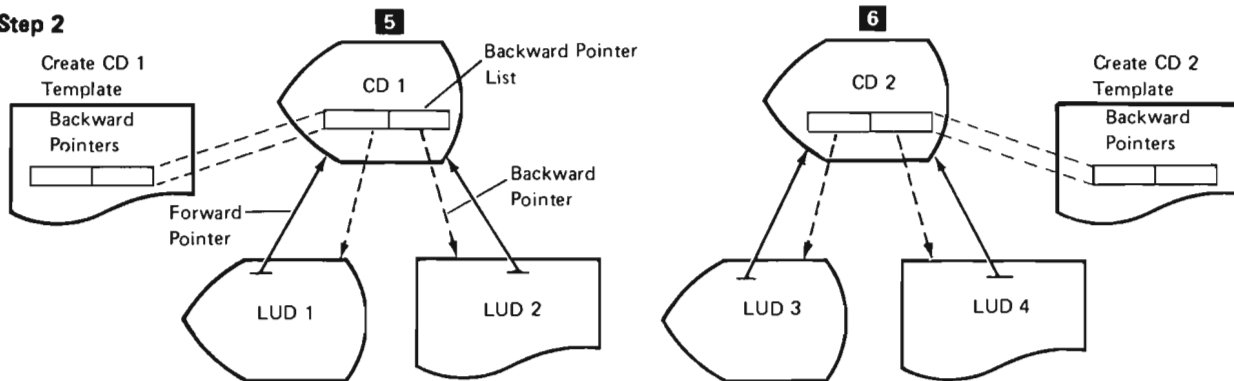
Creating source/sink objects out of sequence:

Step 1	Operation	PTRs Supplied by User	Resulting Action by System
	1 CRTLUD (1)	None	LUD 1 created
	2 CRTLUD (2)	None	LUD 2 created
	3 CRTLUD (3)	None	LUD 3 created
	4 CRTLUD (4)	None	LUD 4 created
Step 2	5 CRTCD (1)	PTR to LUD 1 PTR to LUD 2	CD 1 created, forward pointers to CD 1 inserted into LUD 1 and LUD 2
	6 CRTCD (2)	PTR to LUD 3 PTR to LUD 4	CD 2 created, forward pointers to CD 2 inserted into LUD 3 and LUD 4
Step 3	7 CRTND (1)	PTR to CD 1 PTR to CD 2	ND 1 created, forward pointers to ND 1 inserted into CD 1 and CD 2

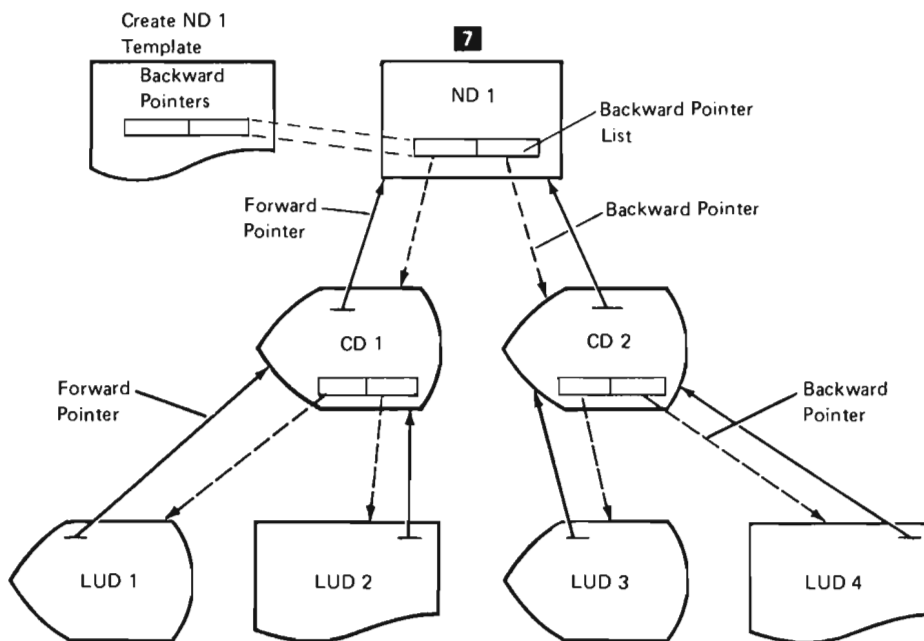
Step 1



Step 2



Step 3



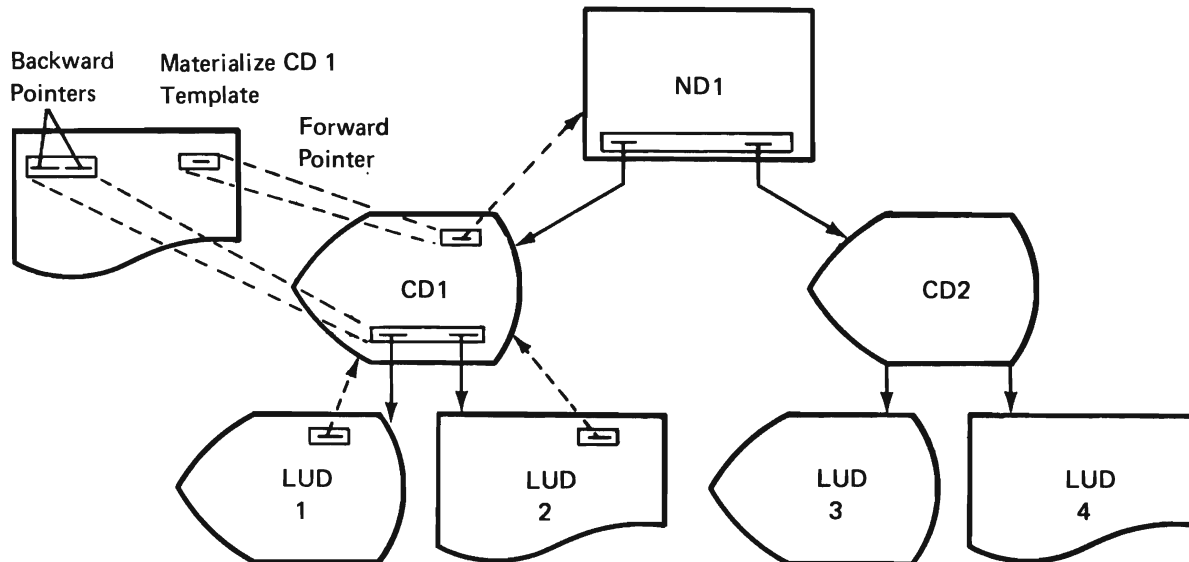
Legend:
 - - - - -> Pointer supplied by user
 - - - - -> Pointer generated by system

Updating a Communications Station

This example shows how to destroy and re-create CD objects to reflect a new capability on a station in the network. Assume CD 1 was updated to provide new features or capabilities (with which this example is not concerned). It is not necessary to destroy LUD 1 and LUD 2 before destroying CD 1 as shown in this example.

Operation	PTRs Supplied by User	Resulting System Action
Step 1 MATCD (1)	None	Provides materialization of contents of CD 1 including list of backward pointers to LUD 1 and LUD 2.

Step 1:

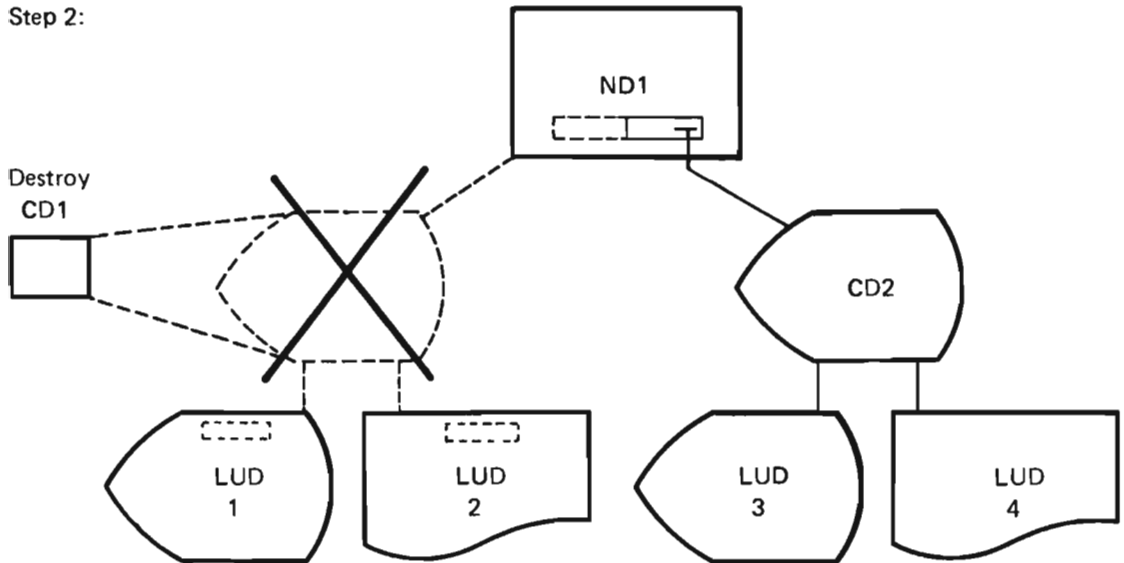


Legend:

- > Pointer is supplied by user
- > Pointer is supplied by system

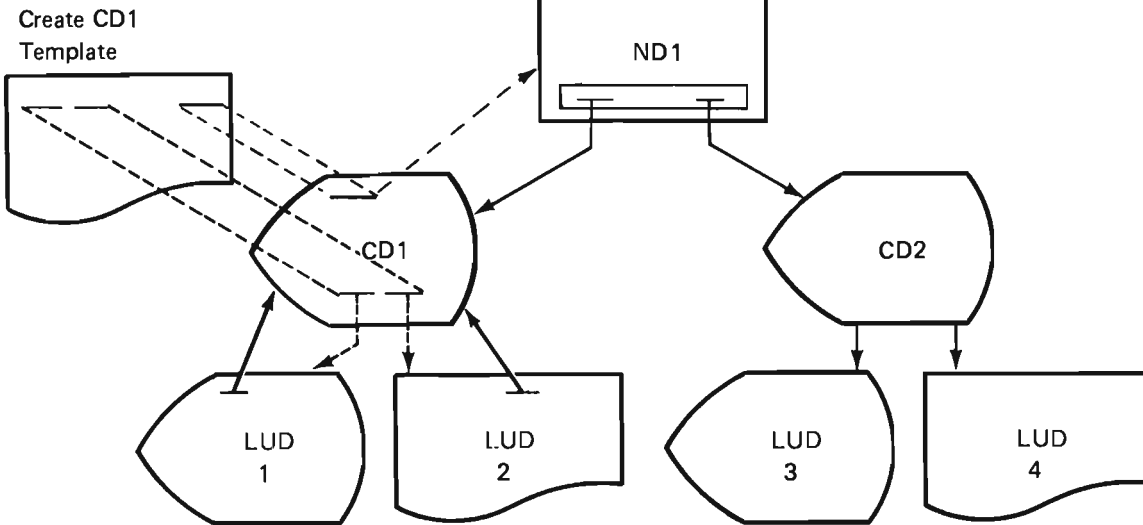
Operation	PTRs Supplied by User	Resulting System Action
Step 2 DESC (1)	None	Destroy CD is completed. Forward pointers within LUD 1 and LUD 2 are deleted. The backward pointer in ND 1 is deleted.

Step 2:



Operation	PTRs Supplied by User	Resulting System Action
Step 3 CRTCD (1-New)	PTR to ND 1 PTR to LUD 1 PTR to LUD 2 from MATCD(1) template	CD 1-New is created using object lists obtained from MATCD(1). ND 1 backward list is updated to include CD 1-New. Forward pointers in LUD 1 and LUD 2 are updated.

Step 3:



Legend:

- > Pointer is supplied by user
- > Pointer is supplied by system

Activation of Switched Networks

The communications network is defined via system objects ND (network description), CD (controller description), and LUD (logical unit description). These objects are created and modified to reflect the unique features, characteristics, and capabilities of each component of the network.

An ND exists for each of the communications line attachments on the system. The CD is associated with each physical station that is attached to a communications line regardless of whether the link is a nonswitched line, a switched network connection, an X.25 permanent virtual circuit, or an X.25 switched virtual circuit. An LUD must exist for each logical unit that can be addressed through the communications network. Some devices have a single logical unit and, therefore, a one-to-one relationship exists between a CD and an LUD. In other cases, a single control unit (represented by one CD) may be the attachment station for multiple logical units, such as interactive terminals or printers. In this situation, many LUDs would be chained off a single CD.

The user controls which components of the communications network are available for use by issuing the Modify ND, CD, or LUD instruction to vary on those components that are explicitly made accessible to other users. Any component not in the varied on state cannot be allocated or used in any way until a Modify instruction is issued by an authorized user to vary on that component.

Vary On and Switched Enable (ND)

The first step in activating the communications network is to use the Modify ND instruction to vary on all the communications lines that may be used and to enable those communications lines that have switched support.

When the attachments for nonswitched lines and local loops are activated, the line is prepared for transmitting to the attached devices. When the ND associated with a communications attachment configured for switched network support is in the enabled state, the attachment is activated and made ready to accept incoming calls.

Vary On CD

The second step is to use the Modify CD instruction to vary on all of the stations that are to be used on this communications network.

Whether the varied on state for the CD is logical or physical depends on the attachment method used for this control unit. If the CD represents a station on a nonswitched line or an X.25 permanent virtual circuit, the physical connection is activated and initial contact with the station is established. In addition, for SNA devices requiring an ACTPU (activate physical unit) message, the MSCP (machine services control point) sends the ACTPU message establishing the MSCP to PU (physical unit) session.

If the CD represents a control unit designed for communications via a switched network or an X.25 switched virtual circuit, the CD is marked as being in the varied on pending state with completion of the vary on sequence to occur later when the switched connection is established.

Vary On LUD

Finally, the Modify LUD instruction is used to vary on the logical units that are eligible for use. Again, if an SNA LU (logical unit) is attached to a station on a nonswitched line or local loop and if the LU requires an activate logical unit message, the MSCP sends the activate logical unit message to establish the MSCP-to-LUD session. It is this MSCP-to-LU session on which unformatted system services requests (for example, logon) can flow as an unsolicited incoming message (as far as the user is concerned).

However, if the LU is attached to a controller in a switched network and the connection has not been established, the LUD is marked as being in the varied on pending state with completion of the vary-on sequence to occur later when the switched connection is established.

Dial In (SDLC or BSC)

For dial in devices, all pending conditions exist until an activated switched line attachment accepts an incoming call or until the logical unit and the controller are moved to the varied off state. When an incoming call is completed, the XID (exchange ID) or SSCPID for support of SDLC primary stations protocol is observed. The ID information received (or sent) is used to identify the calling controller. Checks are made to determine that it is a valid call (a CD exists in the system), that this controller is allowed to call in at this time (the CD is in a varied on pending state), and that the controller is allowed to use this particular switched connection (the ND associated with the line is in the ND candidate list of this CD) or the CD is in the eligibility list of the ND (if the BSC protocol is being used). When all of these conditions are met, the pending conditions associated with the CD and attached LUDs are completed. Then the event, CD contact, is signaled. The event-related data contains a system pointer to the CD that dialed in. The listener of the event must then take the appropriate steps to service the needs of the caller. If the listener does not wish to (or is not able to) service the caller, a Modify CD instruction can be issued to abandon the connection. If there are no listeners for this event or if all the conditions checked are not satisfied, then the switched connection is terminated (the handset is placed in the cradle).

Dial Out (SDLC or BSC)

The system initiates a dial procedure to establish a switched connection at the time that a Modify CD (dial out) instruction is issued to a controller whose CD contains a dial telephone number and is in the varied on pending state. In attempting to complete a dial out connection, the ND candidate list in the CD is again referenced.

The ND candidate list is used in first-to-last sequence. Each ND is checked to determine whether it is in a varied on state, a not-in-use condition, and that the CD and ND are in the same mode (for example, both are in BSC or both are in SDLC). When all of these conditions are satisfied, the dial out procedure is attempted on the corresponding line. If an autodial unit is installed, the call is initiated with no involvement of the operator or a process. However, if the line does not have an autodial unit, then an event (CD manual intervention) is signaled. The event-related data contains sufficient information (for example, telephone number and line attachment) to instruct the operator to perform the dial operation. When the connection is made either manually or by autodial and the ID information has validated the proper connection, all necessary operations are performed as required to satisfy pending conditions, and an event (CD contact) is signaled.

If the manual connection cannot be established by the operator, a Modify CD instruction (abandon connection) should be issued. If an autodial is not successful, the event (CD contact) is signaled indicating unsuccessful contact of the station.

Note: For work station and APPC, the ID is the SDLC XID (exchange ID). For LU1, the ID is the SSCPID associated with the ACTPU control message. For BSC, the ID is the XID entry from the remote XID list.

X.25 Switched Virtual Circuit Call In

For X.25 call in services, all pending conditions exist until an activated X.25 line accepts an incoming call request packet or until the logical unit and the controller description are moved to the varied off state.

When a call request packet is received, the X.25 packet switching data network (PSDN) DTE address and the PSDN connection password are used to identify the calling controller. Checks are made to determine that it is a valid call request (a CD exists in the system), that this controller is allowed to call in at this time (the CD is in a varied on pending state), and that the controller is allowed to use this particular X.25 line (the ND associated with the line is in the ND candidate list of this CD). In addition, if reverse charging was requested in the X.25 call request, the CD is checked to verify that reverse charging can be accepted from the remote controller. Also, the logical link protocol requested in the call request must match the logical link protocol specified in the CD.

When all the previous conditions are met, a positive call request response is sent back to the remote station and the XID (exchange ID) or SSCPID for support of SNA primary stations protocol is observed. Checks are made to verify that the XID or SSCPID of the remote controller matches that of the CD.

When the XID or SSCPID conditions are met, the pending conditions associated with the CD and attached LUDs are completed. Then the event, CD contact, is signaled. The event-related data contains a system pointer to the CD that called in.

The event listener must then take the appropriate steps to service the caller. If the listener does not wish to (or is not able to) service the caller, a Modify CD instruction can be issued to abandon the connection. If all the conditions checked are not satisfied, the switched virtual circuit is terminated (a clear packet is sent to the remote station).

X.25 Switched Virtual Circuit Call Out

The system initiates a call out procedure to establish a switched virtual circuit at the time that a Modify CD (dial out) instruction is issued to a controller whose CD contains an X.25 remote PSDN DTE address and is in the varied on pending state. In attempting to complete a dial out connection, the ND candidate list in the CD is again referenced.

The ND candidate list is used in first-to-last sequence. Each ND is checked to determine whether it is in a varied on state, is not in an inoperative status, and that both the CD and ND are X.25 objects. Also, the default packet size and the maximum PIU size in the CD must not be greater than the maximum packet size or the PSDN maximum PIU size in the ND. Furthermore, the default window size in the CD must not be greater than 7 if ND indicates a modulo 8 X.25 network and a switched logical channel must be available to call out on. When an ND is found that satisfies all of these conditions, a CALL REQUEST packet is sent out on the X.25 PSDN. When a positive response to the call request is received, the ID information must be validated. The CD contact event is signaled when the ID information is validated. If the call request is not successful, an event (X.25 Call Request rejected) is signaled.

Note: For work station and APPC, the ID is the XID (exchange ID). For LU1, the ID is the SSCPID associated with the ACTPU control message.

Summary: The procedure for using the switched communications network includes the following steps:

1. Vary on ND
2. Vary on CD
3. Vary on LUD
4. Dial out CD (dial out network only)
5. Listen for CD contact event
6. Listen for LUD contact event
7. Activate session
8. Request I/O (perform an SNA bind sequence or BSC bid sequence)
Request I/O (data operations)
Request I/O (perform an SNA unbind sequence)
9. De-activate session
10. Abandon connection (CD)
11. Vary off LUD
12. Vary off CD
13. Vary off ND

Note: Calls from devices that dial in are accepted at any time following step 2; the call must be completed before step 6 can be initiated. Calls initiated by the system, (dial out) are made after step 4 and must be completed before step 6 can begin.

Session State Changes

The Modify LUD instruction, when used to change a session state for that LUD, is a synchronous instruction; the Request I/O instruction is asynchronous. The significance of this contrast is that any number of Request I/O instructions may have been issued (and executed) without having those requested operations completed; rather, they are simply scheduled to be processed as if they were enqueued on a process queue. However, only a single Modify LUD instruction can be outstanding at any one time. For example, a Modify LUD (quiesce) remains in an execution state until all request I/O operations outstanding to the session are completed normally and all corresponding feedback records are posted to the source/sink response queue. Only after all of this is completed can the Modify LUD instruction be completed. The session state changes through a Modify LUD instruction are synchronous to ensure that all dispositions of request I/O operations are completed by some determinate time, specifically the point when the Modify LUD instruction completes.

Limitations

Because there are instances when the request I/O operation is stopped, there are limitations on when the synchronous session state changes can be allowed to proceed. A deadlock situation could result if the state changes were to be allowed because they are dependent on completion of the stopped request I/O processing. The two defined cases of stopped request I/O operations that offset the Modify LUD session state change commands are operator intervention and terminating error feedback record.

Operator Intervention: After signaling an operator intervention required event, processing of subsequent requests is suspended until the intervention is resolved or until a Reset Session command is received.

Terminating Error Feedback Record: After indicating a terminating error, further processing of request I/O operations is suspended until a Request I/O Continue instruction or a Reset Session command is received.

The allowable cases are summarized in the following table. The rejected commands result in a resource not available exception to the Modify LUD instruction.

Modify LUD Command	Operator Intervention	Terminating Error
Suspend	Rejected	Allowed
Quiesce	Rejected	Rejected
Reset	Allowed	Allowed
De-activate	Rejected	Rejected—if LUD is active Allowed—if LUD is suspended or reset

Request I/O Operations—Error Recovery Examples

These examples show various error recovery techniques for different conditions that can exist during request I/O operations to an active session on an LUD.

Operator Intervention Required Event

When this event is signaled, it indicates a condition, which can be corrected by intervention at the I/O device. The type of intervention required is identified by the event-related data that is provided when the event is signaled. The definitions of this data and the circumstances under which this event is signaled are defined for each device supported on a system in model dependent documentation.

Some examples of device conditions that can cause this event to be signaled and the required action are listed in the following chart:

Device Condition	Required Action
Device not ready	Start device
Hopper empty	Reload input
Out of forms	Reload printer
Stacker full	Empty stacker

When the operator intervention required event is signaled, processing is suspended on the request I/O operation that experienced the condition. All subsequent requests that may have been issued remain on the queue to await processing. Additional Request I/O instructions are accepted and scheduled for processing but also queued. Normal processing of the requests resumes as soon as the device status indicates that the device condition is corrected and the device is once again operational (presumably as a result of the manual intervention). If manual intervention is not possible, a Modify LUD (reset session) instruction must be issued. This instruction causes all suspended request I/O operations to be terminated abnormally and feedback records for each of these operations posted to the source/sink response queue.

Terminating Error Conditions

Whenever the machine has completed a request I/O operation with a terminating error indicated in the feedback record, all outstanding requests are suspended, and no further processing takes place until certain actions occur at the machine interface. A Modify LUD (reset session) instruction or a Request I/O (continue) instruction cause processing to be restarted. Normal Request I/O instructions can be issued after a terminating error condition and, as in the previous example, they are scheduled but processing on them will not begin.

When a Request I/O (continue) instruction is issued after a terminating error condition, normal processing resumes immediately, and a feedback record for a continue-response is posted to the response queue. This procedure can be used to resume processing if the user chooses to ignore the terminating error. However, this procedure can be used also as a method of recovery if the user diagnoses the cause of the terminating error, makes the necessary correction, and reissues (with a higher priority key and prior to issuing the Request I/O (continue)) the Request I/O (normal) instruction that experienced the terminating error. The reissued Request I/O instruction will be the first I/O instruction to be processed after the system is restarted with the Request I/O (continue) instruction.

A Modify LUD (reset session) instruction issued after a terminating error causes all suspended request I/O operations to be abnormally terminated. The corresponding feedback records are posted to the response queue, and the session on the LUD is changed from the active state to the inactive state. Then, no further Request I/O instructions are accepted. This method can be used to recover from the terminating error condition when a total restart is desired.

A summary of the Request I/O instructions, which are allowed in each of the recovery states mentioned in the previous examples, is shown in the following table.

Command	Operator Intervention	Terminating Error
Request I/O (normal)	Accepted	Accepted
Request I/O (continue)	Rejected	Accepted

The rejected cases result in feedback records that indicate the request was not acceptable for the current condition of the machine.



Chapter 7. Machine Support Functions

System/38 Support Functions

System/38 support functions aid in the initialization, operation, maintenance, and servicing of the machine interface. The support functions are:

- Machine attributes
- Machine-to-programming transition
- Terminate machine processing
- Machine check
- Diagnostic and service aids

MACHINE ATTRIBUTES

Machine attributes are those attributes that have the same value and same meaning throughout the entire machine. These attributes are available to all processes within the machine and have meaning independent of the process that materializes or modifies them. Machine attributes can be modified by a user to govern the operation of the machine, or they can be materialized to determine the current status of a particular aspect of the machine. (For more information about the machine attributes and their formats, refer to the *System/38 Functional Reference Manual*.)

MACHINE-TO-PROGRAMMING TRANSITION

The IMPL (initial machine program load) function can be activated by the machine power-on sequence or from the machine console. After the machine has been initialized, the machine-to-programming transition function provides the user with the capability of initiating a process from the data that exists either on the primary load/dump device or in auxiliary storage. For more information about the IMPL function, refer to the *System/38 Operator's Guide*.

TERMINATE MACHINE PROCESSING FUNCTION

The TMPF (terminate machine processing function) performs any of the following options when machine processing is terminated:

- Enters the machine into the check stopped state.
- Destroys all processes and turns off the machine power supply.
- Suspends all processes and enters the machine into the check stopped state.

The process that invokes the TMPF via the Terminate Machine Processing instruction must have special authority to destroy all processes within the machine. The TMPF is invoked whenever the Terminate Machine Processing instruction is issued.

When the process that issued the Terminate Machine Processing instruction is terminated, all machine resources are returned to the machine.

Once machine processing is terminated or suspended, it can only be reactivated through machine initialization.

An optional function performed by the TMPF is that of turning off the machine power. This option is specified in the Terminate Machine Processing instruction. If the machine power supply is not turned off, the last function performed by the TMPF is that of putting the machine in the check stopped state.

Another optional function performed by the TMPF is that of regenerating the internal supply of machine addresses. The supply of addresses used for only temporary objects or both temporary and permanent objects can be renewed through this function.

The suspend option does not destroy the processes, but keeps them in an internal state to provide information for diagnostic purposes.

The primary power supply can be turned off not only as the result of a process request, but also as the result of some action external to the machine. For example, an abnormal machine power off can occur as a result of a power fault.

The Terminate Process instruction is used to destroy one or more processes with the procedure consistent during the destruction of each process.

MACHINE CHECK FUNCTION

The MCF (machine check function) provides a mechanism whereby machine malfunctions are reported by a machine check exception, a machine check event, or both.

The machine check function is provided to:

- Minimize the possibility of the machine becoming inoperative as a result of a machine malfunction.
- Initialize recovery for instructions causing machine malfunctions when possible.
- Record machine malfunctions internally within the machine.
- Signal machine malfunctions to the machine interface.

Because of its recovery capability, the machine check function minimizes the possibility of the machine becoming inoperative. Only the process that incurs an unrecoverable machine check is terminated (destroyed); all other active processes are not terminated unless the cause of the machine check is determined to jeopardize further execution.

If possible, the machine check function recovers from machine malfunctions. This dynamic recovery capability aids in minimizing the possibility of the machine becoming inoperative.

All machine malfunctions are logged internally if the machine is still operable. The logged malfunctions can be accessed by using the concurrent diagnostic service functions.

A machine malfunction is reported to the machine interface as an exception or an event.

Machine Checks

Machine checks are machine malfunctions from which recovery may or may not be possible. A program monitors a machine check exception or event. As a response to the exception or event, a program can perform some user-defined recovery functions, such as keeping a record of the machine malfunction or destroying the process that incurred the machine malfunction.

Machine checks fall into two categories:

- Soft (recoverable) machine checks
- Hard (unrecoverable) machine checks

All active processes are deactivated when a machine malfunction occurs. The machine determines whether the malfunction is recoverable; if so, the machine saves the recovered machine-related data in its internal storage, activates all processes that were de-activated, and signals the machine check event.

If an unrecoverable machine check occurs because of a System/38 instruction, the machine saves machine-related data in its internal storage, reactivates all processes that were active when the machine check occurred, and signals a machine check exception to the process that incurred the machine check. Execution of the instruction is not completed, and the exception can be handled by the user in the same manner as any other exception.

The service aid functions are used to retrieve the machine-check data saved by the machine.

The machine check exception is signaled to inform a process that it has incurred a permanent machine malfunction. The process determines whether it should terminate itself or perform some recovery operation as a result of the permanent machine check. The machine check exception and related data are defined in the *System/38 Functional Reference Manual*.

Each process that wants to be notified of an unrecoverable machine check that occurred in an instruction it was executing must monitor the machine check exception.

The machine check event is also signaled when a machine check occurs. Any process can monitor a machine check event if it wants to be notified of any machine checks occurring within the machine. The machine check event-related data contains information about the nature and severity of the machine check and referencing the process incurring the machine check.

Machine check event-related data is described under *Event Specifications* in the *System/38 Functional Reference Manual*.

DIAGNOSTIC AND SERVICE FUNCTIONS

The diagnostic and service aid functions provide the service representative with the capability of servicing the machine. These functions can be activated via the Request I/O instruction. Additional information about the diagnostic and service aid functions is contained in the *System/38 Diagnostic Aids Manual* and the *System/38 Service Guide, SY31-0523*.

Machine Observation Functions

Machine observation functions observe the activity of the machine at the machine interface level. Also, these functions are used as a tool to aid in problem determination and to allow observation of machine execution to distinguish its anticipated operation from its actual operation.

Machine observation is explicitly provided through use of the Trace and Materialize instructions and implicitly provided through use of the inherent facilities. Trace instructions are available to monitor the execution of programs and to observe System/38 instructions. Machine activity is observed by monitoring machine events that have event descriptions.

The observation functions provided at the machine interface level are:

- Tracing the execution sequence of a process by monitoring the Call and Return instructions
- Tracing the execution sequence of a program by monitoring instruction execution and program object references
- Tracing the reference to or modification of system objects by using the occurrence of machine events and user-defined events
- Materializing the addressability of pointers

OBSERVATION FUNCTIONS

Observation functions include inherent machine observation functions, trace functions, and materialization capabilities.

Inherent Machine Observation Functions

Certain System/38 instructions monitor machine activities by monitoring the machine events (machine events are defined by event descriptions). Event monitors can monitor the creation and destruction of processes, queues, and other objects. For example, by monitoring the time-of-day clock it is possible to periodically start a machine observation sampling routine.

Trace Functions

A process or a program can be monitored by a monitoring program that specifies the expected event during the execution of the monitored process or program. The following types of trace functions are allowed:

- The Trace Invocation instruction generates an event whenever control is passed from an identified invocation to invoke another program or when control is given up by an identified invocation and control returns to a preceding invocation.
- The Trace Instructions instruction generates an event each time a System/38 instruction is executed by a monitored program.

INHERENT MACHINE OBSERVATION INSTRUCTIONS

System/38 provides programs that monitor and observe events. You may want to use these programs to inform a process when an event occurs. The event could be, for example, a change in the value of an object. Some events can be monitored when:

- A process is initiated or terminated
- A message is enqueued or dequeued
- A limit is reached on a queue
- A time interval is expired

This list shows only some of the events that can be monitored. For more information about event monitors, refer to *Event Management* in Chapter 4.



TRACING

Tracing provides a record of the execution of a program and exhibits the sequences in which the instructions were executed.


The types of tracing provided by System/38 instructions are instruction and invocation. The Trace instructions are:

- Trace Instructions
- Trace Invocations
- Cancel Trace Instructions
- Cancel Invocation Trace

The Trace Instructions instruction identifies a specific set of instructions in a program and when one of those instructions is executed, the instruction reference event is signaled.

The Trace Invocations instruction causes an invocation reference event to be signaled whenever a specific invocation gives up control to either invoke another program or return to a preceding invocation. When an invocation is being traced, the following conditions cause the invocation reference event to be signaled:

- The invocation calls another program.
- The invocation transfers control to another program.
- The invocation is interrupted to handle an event.
- An exception is signaled to the invocation and an external exception handler is invoked.
- The invocation returns to a preceding invocation.
- The invocation terminates processing of an exception and through the Return from Exception instruction passes control to a preceding invocation.
- The invocation is terminated in order to terminate a process phase.
- An invocation exit program is given control.



During the monitoring of these events, it is possible to examine and to modify data in the executing program and return to that program to continue executing with the changed data.

MATERIALIZE INSTRUCTIONS

Materialize instructions can obtain the current values and attributes of system objects. System objects can be materialized into template form by using specific materialize instructions.

The type of object referenced by a system pointer can be determined by using the Materialize System Object instruction. This instruction provides the name, type, owner's name, and other general information about any system object. Each instruction and materialization is described in the section dealing with the object it materializes.

The Materialize Pointer instruction can be used to materialize the current addressability contained in a pointer.

The Materialize Pointer Locations instruction determines which locations within a space contain pointers and provides a bit mapping of their relative location.

The Materialize Invocation instruction allows the program objects not stored in the program's static or automatic storage to be materialized. This information includes the current status of an exception description and the current addressability of internal or external parameters. The object to be materialized is identified by the ODT (object definition table) address and the program invocation number within the process.

The Materialize Invocation Entry instruction can be used to materialize the attributes of a specific invocation entry within the process that issues the instruction.

The Materialize Invocation Stack instruction can be used to materialize the current invocation stack within a specific process.

Recovery Functions

System/38 provides for integrity of system operation and content. However, unexpected failures may occur within the machine and on the part of its users. Because of this, facilities are provided to protect against irrecoverable loss, detect abnormal situations and contain their impact, and to provide recovery mechanisms when failures occur.

RECOVERY CAPABILITIES

The recovery capabilities can be divided into two categories: those related to the data base and those that are general system capabilities.

Because the data base represents the most critical user resource in the system, various recovery capabilities provide the user with the means to protect and recover certain portions of the data base.

Data Base Recovery Capabilities

System/38 instructions provide the following data base recovery capabilities.

Ensure Data Space Entries

In a virtual storage system such as System/38, recent changes to data in main storage may exist and yet these changes may not be reflected in auxiliary storage. If this condition exists, and a power failure or other abnormal system termination causes loss of the data in main storage, the changes will be lost. The Ensure Data Space Entries instruction can help protect against such a loss by ensuring that all changes made via the specified cursor up to that time are recorded in auxiliary storage. This instruction should be used at meaningful intervals in an application to reduce the quantity of critical data that is subject to loss.

Any changes made subsequent to the instruction may or may not be reflected in auxiliary storage.

Forced Write Option

The forced write option on the Set Cursor, Insert Data Space Entry, and Insert Sequential Data Space Entries instructions assures that individual changes to the data base are protected against loss of main storage contents.

Insert Sequentiality

If multiple users (cursors) are inserting (adding) to a file and one user ensures his entries, then all inserts prior to his last entry are also ensured. Updates by other users, however, are not ensured.

Certain types of malfunctions cause a loss of main storage data—for example, a loss of external power. When this type of failure occurs, all data spaces are examined for internal consistency. This procedure enables data base management to guarantee the following attributes of each data space.

- If only insert operations have been performed against the data space, and if the size of the entries within this data space is equal to or less than 512 bytes, all recovered entries now residing in the data space are complete and in sequence.
- Unensured inserts can be lost from the end of the data space, but no ensured inserts can be lost.
- All ensured updates are complete. Unensured updates might exist in a variety of recovery states (for example, not updated at all, partially updated, or completely updated). In each state, however, the data space entry can still be retrieved.
- All ensured deletes are complete. Unensured deleted entries might, in some instances, reappear as retrievable entries.

Object Recovery List

Following abnormal system termination, the object recovery list can be obtained through use of the Materialize Attributes instruction (the machine initialization status record must be requested). This list identifies the data base objects that were detected during IPL as damaged or potentially damaged (for example, the data spaces active at the time of the abnormal system termination and data space indexes invalidated due to abnormal system termination). The object recovery list may also identify damaged data base objects following a normal termination.

Invalidation

Following an abnormal system termination, all data space indexes which might not be consistent with the data spaces they address are invalidated and identified in the object recovery list. These indexes may then be rebuilt by using the rebuild option of the Data Base Maintenance instruction.

File Positioning

Cursors can be positioned to any desired location to begin processing. This capability is useful in case operation on a file is unexpectedly interrupted during processing and must be resumed at an arbitrary point.

System Recovery Capabilities

The following system recovery capabilities are provided.

Ensure Object

The Ensure Object instruction ensures that prior changes to an object residing in main storage are not lost due to an abnormal system termination.

Automatic Ensure

Changes made to certain critical objects are automatically protected against abnormal system termination. This includes addressability changes to contexts, ownership changes to user profiles, structural relationships between data spaces and data space indexes, objects loaded by the load/dump function, and source/sink configuration changes. All created objects are also automatically ensured.

Load/Dump

The capability is provided to dump certain objects to an external media and then reload these objects from this same external media via source/sink management. This capability provides protection from catastrophic damage within the system and also provides the ability to transfer objects to other systems.

Abnormal Termination Indicator

Following IPL, an indicator is available via the machine attributes. The indicator indicates whether or not the previous termination of the machine was normal.

Damaged Contexts

A damaged or destroyed machine context is implicitly rebuilt or recreated when an IPL is performed.

A damaged permanent context that was created with the rebuild recovery option is implicitly rebuilt when an IPL is performed. The Materialize Context instruction is used to materialize the rebuild recovery option for a context.

Damage

Abnormal occurrences within the machine can result in damage to system objects or the operational capability of the machine itself. Following abnormal termination, the machine attempts to restore its operational capability. During the restore operation, certain operations and relationships are designed to be tolerant of damage that has occurred (this includes both machine operations and instruction processing).

When instruction processing cannot be properly completed, an attempt is made to nullify the processing performed by the instruction or to isolate the damage.

Object Isolation

When it is possible to determine that abnormal instruction operation is due to damage to an object, that object is placed in the damaged state. Then, damage exceptions and events are generated. When an object is in the damaged state, the Destroy, Lock, Unlock, Transfer Lock, Materialize System Object, Rename, and Modify Addressability instructions can operate on it. Attempts by other instructions to operate on a damaged object may result in a damaged object exception.

Process Isolation

Unexpected occurrences are isolated to a process whenever possible and, if not handled by the process, only the process is terminated.

Reclaim Lost Objects

When damage occurs within the machine, especially abnormal machine termination, the ownership of objects (as determined by user profiles) can be lost. Such objects are referred to as lost objects. Also, space allocated to severely damaged or destroyed objects may not be properly released. The Reclaim instruction releases space that cannot be associated with a valid object, and also returns a list of lost objects. The Transfer Ownership instruction can then be used to restore object ownership to a user profile.

access group: A system object that is a collection of other system objects, which are transferred to/from auxiliary storage as a group. The access group is used to improve storage management efficiency by specifying which system objects are used together.

activation: The allocation of static storage to a program in a process. A program is activated either implicitly on the first call to a program or explicitly with an Activate Program instruction.

active cursor: A cursor that is currently bound to a process and is thereby available exclusively to this process for purposes of accessing entries residing within data spaces. Only one process can activate any given cursor at a time.

activity trail: A record of operations that is used to identify what activities have been done, the order in which they were done, and who performed the activities.

addressability: The ability to obtain a pointer to a system object or to bytes in a space.

adopted user profile: The user profile that owns the program that has been created with the adopt user profile attribute. The adopted user profile supplements the process with its authorities as long as the process is executing that program. See also *propagated user profile*.

advanced program-to-program communication (APPC): Data communications support that allows a System/38 to communicate with other systems having compatible communications support. APPC is the System/38 implementation of the SNA/SDLC LU6.2 protocol. Using APPC, System/38 can start programs on another system, or another system can start programs on the System/38.

after-image: The image of a record in an object after the data has been modified by a write or an update operation. Contrast with *before-image*.

American National Standards Institute: An organization sponsored by the Computer and Business Equipment Manufacturers Association for the purpose of establishing voluntary industry standards. Abbreviated ANSI.

ANSI: See *American National Standards Institute*.

APPC: See *advanced program-to-program communication*.

apply: For journaling, the process of placing after-images of records into an object. The after-images are recorded as entries in a journal.

argument list: A program object that provides a means of transferring object addressability from an invoking program to an invoked program. It contains a list of ODT references that specify the objects whose addressability is to be passed and the order in which the arguments are to be associated with their corresponding parameters. See also *parameter list*.

arithmetic instructions: Instructions designed primarily to compute numeric results, for example, operations of addition, subtraction, multiplication, and division.

associated space: A space allocated as part of a system object. This is a byte addressable region of storage that is addressed through ODT entries, space pointers, or data pointers.

atomic: Indivisible. An operation or instruction is atomic if its action appears to be instantaneous.

attached: Pertaining to a journal space that is connected to a journal port and is receiving journal entries for that journal port.

attribute: Information that describes the characteristics of system objects or program objects.

authority: The right to access system objects, resources, or functions.

authorization: The process of giving a user either complete or restricted access to a system object, resource, or function.

automatic: One of the attributes of a program object indicating that it is to be mapped into the storage allocated at every program invocation.

auxiliary storage: All addressable storage space other than main storage. Auxiliary storage is located on the system nonremovable disk enclosures. Unlike main storage, auxiliary storage can retain data while machine power is off.

back out: To remove changes from an object in the inverse chronological order from that in which the changes were originally made.

base: The number system in which an arithmetic value is represented.

before-image: The image of a record in an object before the data has been modified by a write, an update, or a delete operation. Contrast with *after-image*.

bias: In binary floating-point storage formats, the constant value that, when added to the signed exponent of a number, produces a non-negative biased exponent. The bias for short format is 127 and for long format is 1023.

biased exponent: (1) In binary floating-point storage formats, the non-negative sum of the signed exponent of a binary floating-point number and a constant value (bias). (2) A value between the maximum and minimum field values that is used to represent the signed exponent of a normalized binary floating-point number. The range of biased exponent values is 1 through 254 for the short format and 1 through 2046 for the long format. Contrast with *signed exponent*.

binary digits: The numbers zero and one, which are used to represent a value in the numbering system that has 2 as its base.

binary floating-point number: (1) In CPF the form of a numeric value that contains a signed significand and a signed exponent. The number's numeric value is the signed product of the number's significand and 2 raised to the power of the number's exponent. (2) Internally, the conceptual form of a numeric value that contains a signed significand and a signed exponent. The number's numeric value is the signed product of the number's significand and 2 raised to the power of the number's exponent. Contrast with *long format* and *short format*, which are used to represent binary floating-point numbers in storage. A binary floating-point number can be a normalized number, a denormalized number, or a signed zero.

binary floating-point value: One of the set of conceptual values supported for binary floating-point operations. The set of values supported is composed of binary floating-point numbers, infinity, and not-a-numbers.

binary point: The point that separates the integer digits from the fraction digits in the numbering system that has 2 as its base, similar to decimal point.

binary synchronous communications: A flexible form of line control that provides a protocol for communication between two stations. Abbreviated BSC.

binding: The explicit or implicit resolving of addressability by assigning the addressability for an object to an instruction operand or pointer that refers to the object.

bit: A unit of computer information equivalent to the result of a choice between two alternatives (such as on-off, 0-1, or yes-no).

Boolean instructions: Instructions that are designed to perform Boolean or logical operations on byte strings.

branch form: An optional form of a standard format instruction that has an operation code extender field and branch target operand(s). The operation code extender field indicates the branch condition(s) to be considered, and the branch target operand(s) indicate the instruction(s) to be branched to if the condition exists.

branch point: A program object that defines the target instruction in a program's instruction stream for some branching instruction.

branching instructions: Instructions that optionally change the sequence of program execution based on some computational operation.

BSC: See *binary synchronous communications*.

byte: A sequence of eight adjacent binary digits that are operated upon as a unit and that constitute the smallest addressable unit in the system.

CD: See *controller description*.

commit block: A system object that records and holds the changes made to an object under commitment control.

commit cycle: The commit processing that occurs after a start commit entry (but before a commit or decommit entry) is placed in a journal space.

commit identifier: A 4000-byte area in the commit block object in which user-supplied data is placed when the Commit instruction is performed.

commit transaction boundary: For commit, a transaction boundary exists when there are no data base changes that can be committed or decommitted.

commitment control: An object is under commitment control when the commit function is being used to record the changes made to the object.

communications device: A device attached to the system via a communications line through which data can be transmitted to and from the system.

communications system: All of the elements in all of the nodes supporting user-to-user communication. SNA defines the logical structure, formats, protocol, and operation sequences among elements of the communications system.

comparison instructions: Instructions that are designed to test the relationship between items of data.

compound ODT reference: A form of instruction operand that is defined by using references to a base operand and one or more suboperands. Several types of compound references are possible to allow explicit pointer basing, subscript references and substring references.

connection point manager: The SNA (systems network architecture) component that provides a common mechanism by which session control, network control, and NAUs communicate with their corresponding elements through the communications network. The units of information that the connection point manager receives from the NAUs (network addressable units), session information to construct the transmission headers and request units. Abbreviated CPM.

constant data: A program object that defines a scalar with fixed attributes and value.

constrained: An attribute of a program indicating that the index and length values used in a reference to a vector or a string are to be checked to assure that the reference is not outside the range of the program object.

context: A system object that contains addressability to system objects by name. It is used in system pointer resolution to obtain system pointers to system objects.

controller description: A system object that defines and describes a device controller or communications station. There is one controller description for each device controller or communications station on the system. The controller logically represents a physical I/O controller to the system. Abbreviated CD.

convert: The process of changing the value of a scalar to correspond to a different set of attributes. For example, a decimal scalar could be converted to a binary scalar.

cursor: A system object used with the data base facility. A cursor provides a path to access a data base, performs field mapping and conversion, and retains information about the current status of its use by a process. See also *active cursor*.

data base: A structure of files and indexes that holds data and the relationship among the data. In System/38, a data base is composed of a combination of the following system objects:

- Data space—file of entries (records)
- Data space index—provides logical ordering of entries in data spaces
- Cursor—path to entries in data spaces

data circuit-terminating equipment (DCE): The equipment installed at the user's premises that provides all the functions required to establish, maintain, and terminate a connection, and the signal conversion and coding between the data terminal equipment and the line. Abbreviated DCE. See also *data terminal equipment*.

data pointer: A pointer that provides addressability to a specific byte location in a space and associates scalar attributes with the data addressed.

data space: A system object in which data space entries (records) are stored. Once a data space has been created, new entries can be inserted and existing entries can be updated, retrieved, or deleted.

data space entry: An ordered set of fields (record) that is contained within a data space (file). All entries within a data space have the same number of fields and identical attributes.

data space index: A system object that is used to logically order entries in one or more data spaces.

data terminal equipment: That part of a data station that enters data into a data link, receives data from a data link, and provides for the data communication control function according to protocols. Abbreviated DTE.

DCE: See *data circuit-terminating equipment*.

deadlock: An impasse that occurs when multiple processes form a loop, each waiting for the availability of a resource that will not become available because it is being held by another process that is also waiting in the loop.

default rounding mode: The mode that is put in effect upon the initiation of a process. The default rounding mode is round to nearest.

denormalized number: (1) In binary floating-point concepts, a nonzero number whose significand integer digit has a value of binary zero and whose signed exponent has a value of -126 or -1022. Denormalized numbers provide for a gradual underflow toward zero of numbers representable in a storage format. (2) In binary floating-point storage formats, a denormalized number is represented by an exponent field that contains a reserved value (zero) at the format's minimum and a fraction field whose value is greater than zero. The significand of the number represented has an integer value of zero, which is implied by the storage representation and a fraction value from the fraction field. The sign of the number represented is positive for a sign field value of binary 0 and negative for a sign field value of binary 1. The reserved value of zero in the exponent field indicates that the value of the signed exponent (power of 2) is decimal -126 for the short format and decimal -1022 for the long format.

dequeue: An operation for removing messages from queues. Contrast with *enqueue*.

destination: See result field.

detached: Pertaining to a journal space that is not connected to a journal port and is not receiving entries for that journal port.

DHCF: See *distributed host command facility*.

distributed host command facility: That part of a System/38 that helps to create the communication link between a System/370 terminal and a System/38 application. Abbreviated DHCF.

DTE: See *data terminal equipment*.

dump: To copy the contents of a system object to diskette. The system object is *loaded* by reading it from the diskette and then replacing an existing system object or creating a new object.

editing instructions: Instructions that are designed to allow the programmer to change the format of data items.

eight-byte format: Same as long format.

encapsulation: The process of translating data (such as a program in the form of a template) into an internal and machine usable form. Nonencapsulated system objects are visible in both format and function to the System/38 user. Encapsulated system objects are visible to the user in function, but not in format. An independent index is an example of an encapsulated system object while a space is not encapsulated.

enqueue: An operation for placing messages on a queue. Contrast with *dequeue*.

ensure: Writing all changes made to a system object to auxiliary storage.

entry: See *data space entry*.

entry point: A program object used to define the target instruction in an instruction stream.

event: An asynchronous signal that a process can intercept.

event handler: A program, specified in an event monitor, that is to receive control when the event occurs.

event monitor: A specification of event(s) to be intercepted by a process and the event handler program to be invoked as a result.

exception: The occurrence of a machine or user-defined condition that can be monitored and is directly associated with the execution of a particular function within a process. Exceptions generally represent an abnormality detected in the machine or in a program. Exceptions are signaled to an exception description within the associated process.

exception description: A program object that is used to contain information pertaining to the handling of an exception.

exchange identification: A unique identification of the physical unit that a controller description object represents. Abbreviated XID.

existence: An attribute of a system object that is specified at the time the system object is created. A *permanent* system object exists until it is explicitly destroyed by a Destroy instruction. A *temporary* system object exists until it is explicitly destroyed or until the next machine termination, whichever occurs first.

exponent: A number, indicating to which power another number (the base) is to be raised.

exponent range: In binary floating-point storage formats, the set of integer exponents for normalized numbers that can be represented in a particular format. The representable signed exponent range is decimal -126 through +127 for short format and decimal -1022 through +1023 for long format.

external: One of the attributes of a named data program object indicating that it can be referred to by a program other than the program in which it is defined. Data pointers are used to refer to external program objects.

external entry point: Defines the first instruction to be executed in a program when it is invoked.

feedback record: A message placed on a queue indicating the status of a completed operation initiated by the Request I/O instruction.

field: A portion of a data space entry that serves as the basic unit of data transfer to and from data spaces.

floating-point format: In binary floating-point representation the storage format used to represent a binary floating-point value. See also *long format* and *short format*.

fork character: One or more characters included in the composite keys in a data space index. These characters serve as key modifiers, providing particular ordering of entries with the same key but residing in different data spaces.

format's maximum: In binary floating-point storage formats, the value of 255 (short format) or 2047 (long format) in the exponent field. This value indicates that either a signed infinity (value of the fraction field equals zero) or a not-a-number (value of the fraction field does not equal zero) is represented in the storage format.

format's minimum: In binary floating-point storage formats, the value of zero in the exponent field. This value indicates that either a zero, floating-point value (value of the fraction field equals zero) or a denormalized number (value of the fraction does not equal zero) is represented in the storage format.

forward recovery: The process of recovering a data base file from a known point by restoring a previous version of the file and then applying the changes to that file in the same chronological order in which they were originally made.

four-byte format: Same as short format.

fraction: (1) In binary floating-point number concepts, the value to the right of the binary point in the significand. (2) In binary floating-point storage formats, the value contained in the fraction field.

function check: A type of exception that indicates the malfunction of a specific System/38 instruction.

generic instruction: Those instructions that accept various types of system objects or program objects as operands.

hex: See *hexadecimal*.

hexadecimal: Pertains to a number system with a base of 16. (Valid digits range from 0 through F, where F represents the highest units position-15.) Abbreviated hex.

I/O manager: A programming module (or program) that controls the flow of data and control information to and from an I/O unit. Abbreviated IOM.

I/O port: The system hardware that supports the attachment of I/O devices.

IDL: See *instruction definition list*.

immediate data operand: An operand that contains the data within the instruction. An immediate data operand contains the description and value for either branch targets (instruction number or relative instruction number) or scalar data (binary, character, or bit).

implicit leading bit: In binary floating-point formats, a bit that does not appear in the storage representation of a binary floating-point number. This bit is understood to be to the left of the assumed binary point. See also *significand*.

independent index: A system object that provides for ordering byte data according to the value of the data.

independent process: A process that can exist after its initiator has terminated.

indicator form: An optional form of a standard format instruction that has an operation code extender field and indicator operand(s). The operation code extender field indicates which resulting conditions are to cause the indicator values to be set to the requested values.

inexact result: In binary floating-point operations, a state that occurs when bits of the significand are lost in rounding the intermediate result to the precision of the result field or when a number or infinity is stored as the result of a masked overflow.

infinity: (1) In binary floating-point concepts, a value with an associated sign that is mathematically greater in magnitude than any binary floating-point number. (2) In binary floating-point storage formats, infinity is represented by an exponent field that contains a reserved value at the format's maximum (255 for short format and 2047 for long format) and a fraction field value of zero. The sign of infinity is positive for a sign field value of binary 0 and negative for a sign field value of binary 1.

infinity arithmetic: In binary floating-point operations, the adding, subtracting, multiplying, dividing, and comparing of values that involve infinity values. In infinity arithmetic, negative infinity is less than every number and every number is less than positive infinity.

instruction definition list: A program object used to describe an ordered list of instruction numbers and branch point references. It is used with an index value to accomplish branch table functions. Abbreviated IDL.

instruction stream: A string of System/38 instructions that define an execution time function. The instruction stream is used in conjunction with an ODT to create a program. Each instruction is in operator-operand format with a 2-byte operation code field, an optional 2-byte operation extender field, and from 0 to n 2-byte operand fields.

integrity: (1) The protection of data and programs from inadvertent destruction or alteration. (2) The assurance that a system object is not misused. This assurance is based on associating instructions with specific system object types.

intermediate denormalized floating-point number: In binary floating-point operations, an intermediate unrounded form of the result in which a value that is too small to be represented in the floating-point format of the result has had the significand's digits shifted right (zeros are supplied on the left) and the exponent incremented until the exponent attains the format's assumed value for denormalized numbers (-126 for short format and -1022 for long format).

intermediate result: In floating-point concepts, the normalized number produced prior to the adjustments required to store it in the result field.

invocation: An invocation is the execution of a program. It represents the status of the process after the program is invoked. When one program calls another program, the two programs are said to be in different invocations. The invocation of a program that is called a second time by the same calling invocation is also considered to be a different invocation. Automatic storage is allocated for a program at every invocation.

IOM: See *I/O manager*.

journal: (1) A chronological record of the changes made to a set of data; the record may be used to reconstruct a previous version of the set. (2) To record transactions against a data set so that the data set can be reconstructed by applying transactions in the journal against a previous version of the data set.

journal entry: A record in a journal space. The record contains information about selected actions against an object being journaled.

journal entry-specific data: The user-generated or system-generated data in a journal entry. This data is unique to the operation that generated the journal entry.

journal port: An object through which entries are placed in a journal space when a change is made to an object. The system uses the journal port to record information about the attached journal spaces and the objects being journaled through the journal port.

journal space: The object that contains journal entries.

local work station: A work station that is connected directly to the data processing system (channel) without need for data transmission facilities. Contrast with *remote work station*.

lock: A control applied to a system object (in behalf of a process) that guarantees the ability for a process to perform certain types of operations while prohibiting other processes from performing certain types of operations. The five types of locks are:

- LSRD—Lock for shared read
- LSRO—Lock for shared read only
- LSUP—Lock for shared update
- LEAR—Lock for exclusive use but allow read in other processes
- LENR—Lock for exclusive use with no read in other processes

logical unit description: A system object that defines and describes an I/O device on the system. There is one logical unit description for each I/O device on the system. Abbreviated LUD.

long format: (1) In CPF, in binary floating-point storage format, the 64-bit representation of a binary floating-point number, a not-a-number, or infinity. (2) Internally, in binary floating-point storage formats, the 64-bit representation of a binary floating-point number, a not-a-number, or infinity. The long format is a bit string in which bit zero is the sign field, bits 1 through 11 are the 11-bit exponent field, and bits 12 through 63 are the 52-bit fraction field.

LUD: See *logical unit description*.

machine attribute: Information pertaining to the overall system; for example, date, time of day, and machine configuration.

machine check: A type of exception that indicates a malfunction of the machine.

machine context: A system object implicitly created and maintained by the machine for maintaining addressability to certain types of system objects.

machine interface: The instruction set interface to the System/38 machine. Abbreviated MI.

machine services control point: The machine component that provides services and coordinates the processing of supervisory services.

machine termination: The termination of all processes in the machine with the intent of turning power off or performing an initial microprogram load (IMPL).

magnitude: The relative size of a number.

main storage: The high-speed portion of machine storage used for objects and processes when they are being referred to or when they are being executed. Main storage cannot retain data while machine power is off. Contrast with *auxiliary storage*.

mapping: The reordering, conversion and selection of fields in data space entries when referred to by a program through the use of a cursor.

materialize: The reverse of the process of encapsulation. Materialization is the process of retrieving encapsulated data and presenting it in the form of a template of byte data in a space.

message: A communication sent from one person or program to another. In particular, a message is enqueued on a *queue* by one process and dequeued by another process.

MI: See *machine interface*.

name resolution: (1) The function of resolving addressability to system objects. An unresolved system pointer specifies the symbolic name of a system object. At first reference, an unresolved system pointer is resolved as follows. The machine searches for the symbolic name in contexts until it is found and then sets addressability to the corresponding system object into the pointer, thereby making it a resolved system pointer. The contexts to be searched are contained in the name resolution list. (2) Also, the function of resolving addressability to external program objects defined in programs within a process.

name resolution list: A process attribute that is a vector of resolved system pointers to the contexts that are searched for name resolution. See also *name resolution*.

NaN: See *not-a-number*.

ND: See *network description*.

negative infinity: See *infinity*.

network: The term network has several meanings. A *public network* is a network established and operated by common carriers or telecommunications administrations for the specific purpose of providing circuit-switched, and nonswitched-circuit services to the public. A *user application network* is a configuration of data processing products (such as processing units or work stations) established and operated by users for the purpose of data processing or information exchange; such a network may use transport services offered by common carriers or telecommunications administrations.

Network, as used in this publication, refers to a user application network.

See also *packet switching data network*.

network addressable unit: A port provided for user access to the communication system in SNA. A network addressable unit is either the origin or destination of information units flowing in the communications system. Abbreviated NAU.

network description: A system object that defines and describes an I/O port and communications line for remotely attached I/O devices. The network description logically represents the I/O port to the system. Abbreviated ND.

network port: The system hardware that supports the attachment of communications lines.

normalized number: (1) In binary floating-point concepts, a number whose significand's integer digit has a value of 1. (2) In binary floating-point storage formats, a normalized number is represented by an exponent field that contains a biased exponent. The significand of the number represented has an integer value of 1, which is implied by the storage representation, and a fraction value from the fraction field. The sign of the number represented is positive for a sign field value of binary 0 and negative for a sign field value of binary 1. Note that the exponent field values of 0 and 255 for the short format and 0 and 2047 for the long format are used to indicate the representations of infinity, not-a-number, denormalized number, and signed zero.

not-a-number: (1) In binary floating-point concepts, a value, not interpreted as a mathematical value, which contains a mask state and a sequence of binary digits. The mask state can be either masked or unmasked. The binary digits have no meaning attached to them other than to give the not-a-number a unique value. (2) In binary floating-point storage formats, a not-a-number is represented by an exponent field that contains a reserved value at the format's maximum (255 for short format and 2047 for long format) and a fraction field whose value is greater than zero. The leftmost bit position of the fraction field, bit 9 for the short format and bit 12 for the long format, indicates the mask state. A leftmost bit value of 1 indicates masked, 0 indicates unmasked. The remaining bit positions of the fraction contain the sequence of binary digits associated with the not-a-number. The sign field value does not have a defined meaning. (3) In binary floating-point operations, unmasked not-a-numbers force detection of an exception condition when they are input to an operation. Masked not-a-numbers do not force an exception condition, but are moved into the result field. A not-a-number may represent the result of incorrect combinations of operands in a floating-point operation.

object: A term referring to either a system object or a program object. Objects are referred to by instructions through their operands.

object authority: The right to use a system object. There are eight object authorities:

- Object control—to control existence
- Object management—to control access and use
- Authorized pointer—to allow storing authority in a system pointer
- Space—to control access to the associated space
- Retrieve—to allow retrieving elements
- Insert—to add new elements
- Delete—to remove old elements
- Update—to modify existing elements

Contrast with *lock*.

object authorization: A specification that indicates which system objects a user can access and what rights of use have been granted relative to those system objects. See also *object authority*.

object definition table: A part of a program definition template that defines the program objects associated with the instructions in its instruction stream. Operands of an instruction refer to entries in the ODT. Abbreviated ODT.

object owner: The user profile that owns a permanent system object. The storage occupied by the system object is charged against the owner's storage resource authorization. The owner also retains certain implied authorization rights to the system object.

ODT directory vector: One of the components of the object definition table (ODT). The ODV consists of a series of 4-byte entries. These entries are referred to by the operands of instructions and provide a description of the program object. The ODT entry string (OES) is used to complete the description when it cannot be completely described with the 4-byte ODV entry. Abbreviated ODV.

ODT extender string: One of the components of the object definition table (ODT). The OES contains variable-length entries specifying attributes, initial values, and other items necessary for defining program objects. OES entries are not directly referred to by System/38 instructions, but are referred to via the ODT directory vector (ODV) entries. Abbreviated OES.

ODV: See *ODT directory vector*.

OES: See *ODT extender string*.

operand field: In an instruction, a field specifying the program object associated with a given instruction operand. The field is an index into the object definition table, which describes the program object. Instructions have from zero to four operands.

operand overlap: The situation in which source operands of an instruction share at least a portion of their space storage with the receiver operand of the instruction.

operation code extender field: In an instruction, an optional 2-byte field that follows the operation code field and that further defines the operation to be performed and the instruction format.

operation code field: The first field in an instruction. This 2-byte field defines the basic operation to be performed, basic information about the instruction format, and miscellaneous status information for the instruction.

ordinal entry number: The absolute entry number of a data space entry in a data space. An ordinal entry number is one way to refer to a particular entry in a data space.

parameter list: A program object that provides a means of associating addressability of program objects defined in an invoked program to program objects in the invoking program. It consists of a list of ODT references specifying the program objects that must obtain addressability and the order in which they are to be associated to their corresponding arguments. See also *argument list*.

pointer: A special kind of data contained in a space which is distinguishable from ordinary data by the machine. A pointer can be generated only by specific machine instructions. If the contents of a pointer are altered, the machine no longer recognizes it as a pointer. There are four types of pointers:

- System pointer—addresses system objects
- Space pointer—addresses a byte location in a space
- Data pointer—describes and addresses a byte location in a space
- Instruction pointer—addresses an instruction in a program

positive infinity: See *infinity*.

power: The number of times as indicated by an exponent that a number occurs as a factor in a product.

primary station: In SNA, the station on an SDLC data link that is responsible for control of that data link.

priority: The relative significance of one process to other processes. Priority specifies the relative order of resource allocation when there is competition for a resource.

private authority: The object authority to a system object granted to a specific user profile.

privileged instructions: Those instructions that have the designation of being privileged. Authorization for the use of privileged instructions by a user is defined in each user's profile.

process: An execution sequence (a task) in the machine. Programs are executed in a process in behalf of a user. Processes may communicate with each other through the use of events, messages on queues, or other common data areas.

process control space: A system object used to support the execution of a process and as a means of addressing a process.

processor eligibility definition class: An attribute of a process that defines the relative eligibility of the process to contend for processor time when the process is dispatchable (not waiting) and not otherwise suspended from execution.

program: An executable system object that incorporates both the instruction stream and the data definitions for the instruction operands.

program object: One of two object classifications. Program objects are the operands of program instructions. Program objects are described by object definition table entries and are either contained in spaces or are constant items defined within the program. Contrast with *system object*.

propagated user profile: An adopted user profile whose authority is propagated to other invocations. Propagation is determined by an attribute that is specified when the program is created.

PSDN: See *packet switching data network*.

public authority: The object authority to a system object granted to all users.

PVC: See *permanent virtual circuit*.

queue: A system object consisting of an ordered list of messages that communicate information to other processes.

quiesce: In source/sink operations, the process of completing all outstanding source/sink operations and freeing the device for a new use.

receiver operand: An instruction operand into which the results of the operation are placed. The receiver operand can be the same as one of the source operands.

record: See *data space entry*.

recovery: The actions taken in response to an error or failure.

relative positioning: A method for referring to data space entries in a data space relative to the position of other entries in the data space.

remote work station: A work station whose connection to a data processing system uses modems and common carrier or private data transmission facilities. Contrast with *local work station*.

remove: In journaling, the process of replacing the records in a data space with their before-images as recorded in a journal space.

reserved: A field in a template that must contain binary zeros on input and contains binary zeros on output. Input other than zero may cause an exception or unpredictable results.

reserved values: In binary floating-point storage formats, the exponent field values of 0 and 255 for the short format and 0 and 2047 for the long format that are used to indicate representations of infinity, not-a-number, denormalized number, and zero.

reset session: In source/sink operations, the process of quiescing but without regard for the disposition of outstanding source/sink operations.

response queue: A queue that is used as a container for indicating the results of a requested source/sink operation.

result field: That part of storage in which the result of a calculation is stored.

round to nearest: In binary floating-point operations, the rounding mode that modifies an intermediate result to the nearest representable value. However, if the value of the significand of the intermediate result is exactly halfway between the two representable values, the value chosen is the one whose least significant digit is even.

round toward negative infinity: In binary floating-point operations, the rounding mode that modifies an intermediate result to the representable value that is closest to but no greater than the intermediate result. The result may be negative infinity.

round toward positive infinity: In binary floating-point operations, the rounding mode that modifies an intermediate result to the representable value that is closest to but no less than the intermediate result. The result may be positive infinity.

round toward zero: In binary floating-point operations, the rounding mode that modifies an intermediate result to the representable value that is closest to and no greater in absolute value than the intermediate result. The result may be zero.

rounding: In binary floating-point operations, the process of choosing a representation in the format of the result field for an intermediate result regarded to be of infinite precision.

rounding mode: In binary floating-point operations, one of the four selectable modes in which rounding is performed. The four rounding modes are: round to nearest, round toward negative infinity, round toward positive infinity, and round toward zero.

scalar: A single numeric or character value. For example, a scalar can be a 2-byte binary number but not a pointer. Contrast with *pointer* and *vector*.

SDLC: See *synchronous data link control*.

secondary station: In SNA, any station on an SDLC data link that is not the primary station. It can exchange data only with the primary station; there is no data traffic from secondary station to secondary station.

select/omit: A program may be associated with a data space index that includes or excludes (based on the contents of the entries) data space entries in the index.

session: (1) The period of time during which communication is established between the system and a user. (2) The formal bound pairing that must be established between two network addressable units before their users can communicate.

short form: An optional form of a standard instruction in which one of the first operand acts as both a source operand and a receiver operand.

short format: (1) In CPF, in binary floating-point storage format, the 32-bit representation of a binary floating-point number, not-a-number, or infinity. (2) Internally, in binary floating-point storage formats, the 32-bit representation of a binary floating-point number, not-a-number, or infinity. The short format is a bit string in which bit zero is the sign field, bits 1 through 8 are the 8-bit exponent field, and bits 9 through 31 are the 23-bit fraction field.

sign: In floating-point representation, the leftmost bit of the format is the sign of the significand.

signed exponent: In floating-point operations, the arithmetic representation of the exponent value of the floating-point number. Contrast with *biased exponent*.

signed zero: (1) In binary floating-point concepts, the number zero with an associated sign. It can be thought of as having a significand value of zero and an exponent value of zero. (2) In binary floating-point storage formats, signed zero is represented by an exponent field that contains a reserved value (zero) at the format's minimum and a fraction field value of zero. The sign of zero is positive for a sign field value of binary 0 and negative for a sign field value of binary 1; however, positive zero is always considered equal to negative zero.

significand: (1) In CPF, in binary floating-point, the part of a number that contains the integer and fraction. (2) Internally, in binary floating-point concepts, the part of a binary floating-point number that is composed of binary digits that contain one integer digit to the left of the binary point and one or more fraction digits to the right. (3) In binary floating-point storage formats, the significand is represented in the fraction field, in that the value of the integer digit is implied by the number represented, zero for denormalized numbers and 1 for normalized numbers, and the fraction digits are contained in the fraction field.

simple ODT reference: A single 2-byte operand entry that refers to a program object defined in the ODT.

SNA: See *systems network architecture*.

source operand: An instruction operand containing data to be operated on by the instruction. Contrast with *receiver operand*.

source/sink: Devices capable of originating or accepting data signals to or from a transmission device and the data management components supporting such devices. Source/sink devices include locally and remotely attached, batch and work station devices, but not the internal storage of the system.

source/sink request: The operand of a source/sink Request I/O instruction that specifies the I/O operation to be performed, the characteristics of the data to be used in the operation, and the data to be used in the operation.

source/sink resource: A system resource allocated and deallocated in units described by logical unit descriptions (device descriptions) to the process requiring the resource.

space: (1) The associated space of a system object. This is a byte-addressable region of storage that is addressed through ODT entries, space pointers, or data pointers. (2) A system object that has no functional part but is used only for its associated space.

space pointer: A pointer with addressability to a byte within a space.

static: One of the attributes of a program object. Objects having the static attribute are allocated space and are initiated when the program containing the program object is activated.

string: A linear sequence of bytes such as a character string.

suspend: (1) A system object is suspended if its storage is truncated to a minimum required to maintain its existence in the machine. A suspended system object is not functionally usable until it is *loaded*. (2) A source/sink session is suspended to terminate outstanding I/O operations. (3) A process is suspended if it is made ineligible to compete for processor or main storage resources.

symbolic name: The name of a system object or an external program object. The input to a name resolution function whereby an instruction operand referring to an object by name outside its program is bound to the actual object. System pointers are resolved to system objects; data pointers are resolved to externally defined program objects.

synchronize: (1) To cause to occur at the same time.
(2) To ensure that two objects contain exactly the same information.

synchronous data link control: A discipline for the management of information transfer over the data communications channel. Transmission exchanges can be duplex or half-duplex; the communications channel configuration can be point-to-point, multipoint, or loop. Abbreviated SDLC.

system default not-a-number: (1) In binary floating-point operations, the not-a-number value set as the result for certain combinations of invalid operands. It is masked with a binary digit sequence of all zeros.
(2) In floating-point storage formats, it is represented with a sign field value of zero, and a fraction field value of a leading one bit (to indicate the masked state), followed by all zeros.

system object: A functional part and an associated space part. The functional part of a system object is a construct whose internal format is not visible. Programs, queues, contexts, and user profiles are examples. System/38 instructions are used to perform high-level logical functions on system objects. System objects can optionally contain an associated space. System objects are referred to through system pointers.

system pointer: A pointer that provides addressability to a system object.

systems network architecture: The total description of the logical structure, formats, protocols, and operational sequences for transmitting information units through the communications system. Abbreviated SNA.

template: A contiguous string of byte data in a space organized as a specific data structure typically used for a System/38 instruction operand for source and result data.

temporary object: A system object that is automatically destroyed at machine termination.

unbiased exponent: See *signed exponent*.

unordered: In binary floating-point concepts, the relationship that can exist between two values that indicates that they cannot be ordered according to relative value. The relationship between two values is unordered either when a not-a-number is compared to any value or when infinity in projective mode is compared to any value other than infinity.

user profile: A system object identifying a system user and containing authorization, auxiliary storage limitations, and object ownership information for that user.

vector: A number of scalars or pointers, each of which has the same attributes, located in contiguous bytes of a space.

view: The definition or description of a program object. See *object definition table* and *program object*.

work station: Elements of data processing equipment through which a system's end user has access to a computer as required for the performance of his job (work) at the physical location (station) where he performs job tasks. Examples are display/keyboard devices and printer/keyboard devices. See also *local work station* and *remote work station*.

XID: See *exchange identification*.

X.25: In data communications, a specification of the CCITT that defines the interface to an X.25 (packet-switching) network.



A

- abandon call 4-33, 4-33, G-7, 4-33, 4-39, 4-33, 06-067
- abandon connection 6-67
- abnormal termination indicator 7-7
- abnormal value attribute 2-28
- absolute instruction numbers 2-43
- absolute value 5-15
- access group
 - category 2-1
 - definition G-1
 - description 2-1
 - membership 2-8
 - storage resource function 4-37
- access path to objects 1-1
- access state of an object 4-33
- access to data space entries 5-3
- accessing shared data spaces 5-6
- achieve maximum performance from the LD function 6-64
- action code for return from
 - exception 4-29
- action to take when exception occurs 2-23
- actions specified by exception
 - description 4-24
- activated state of cursor 5-3
- activating
 - a cursor 5-4
 - a session 6-33
 - the communications network 6-78
 - the program 3-36
- activation creation 3-2, 3-36
- activation destruction 3-38
- activation functions 3-35
- activation of switched networks 6-78
- activation, definition of G-1
- active cursor, definition of G-1
- active cursors 5-10
- active or suspended status 4-11
- active session state 6-22
- active state 4-2
- activity trail, definition of G-1
- ACTLU (activate logical unit) control message 6-50
- ACTPU (activate physical unit) session control message 6-50
- adding a data link 6-70
- adding the authorizations held by user profile 2-57
- address of a variably addressed
 - argument 2-40
- address resolution
 - functions 2-30, 2-31
 - object 4-6
- addressability
 - contained in a space pointer 2-27
 - definition G-1
 - modification 2-44
 - provided by pointer data objects 2-18
 - to a newly created object 2-8
 - to a system object 2-5
 - to an object 1-2
- addressing
 - a context 2-31
 - argument and parameter 2-39
 - by generic key 5-14
 - by key 5-14
 - characteristics common to all system objects 2-30
 - each byte in a space 5-37
 - instruction 2-43
 - ODT 2-26
 - pointer 2-26
 - process 2-43
 - space 2-32
 - system object 2-30
 - system objects 2-29
- adopted user profile
 - attribute 4-5
 - authorities 2-47
 - authority verification 2-59
 - definition G-1
 - process authorization 4-5
 - use of 2-47
- advanced program-to-program communications (APPC) 6-45
- advantages of creating source/sink objects in sequence 6-28
- advantages of multiprogramming
 - support 1-6
- after image, definition of G-1
- algebraic key field 5-15
- all object authorities 2-46
- all-object authority 2-49
- all-object authority attribute 2-55
- alter access state of object 4-38
- altering an I/O configuration 6-69
- altering the NRL 2-32
- alternate collating sequence 5-15
- alternate collating sequence on any key field 5-14
- APPC (advanced program-to-program communications) 6-45
- application layer 6-43
- apply, definition of G-1
- applying journaled changes 5-24

- apportioning the main storage resource 4-33
- approach to reducing deadlock situations 4-45
- approximate key addressing 5-15
- argument and parameter, addressing form 2-26
- argument data elements 2-39
- argument list
 - definition G-1
 - entry 2-40
 - interinvocation communications 3-49
 - interprocess communication 4-12
 - reference 2-40
 - use of 2-40
- argument/parameter correspondence 2-41, 3-51
- arguments
 - addressing 2-39
 - interinvocation communications 3-49
 - use of 3-48
- arithmetic instructions
 - definition G-1
 - description 3-25
- arithmetic operations 3-14
- arrange in ascending order, key field 5-15
- arrange in descending order, key field 5-15
- array
 - addressing 2-37
 - attributes 2-20
 - index operations 3-35
 - size entry 2-20
- arrays of elements 2-20
- ascending order with respect to the key field 5-14
- assignable sessions 6-45
- assigning a value to a data pointer 2-27
- associated option value, source/sink 6-28
- associated space portion of an object 1-2
- associated space, definition of G-1
- associating scalar data objects with a space pointer 2-32
- asynchronous
 - event handling 4-19
 - events 6-27
 - initiation step 4-2
- atomic, definition of G-1
- attached journal spaces, load/dump 6-65
- attached, definition of G-1
- attribute
 - binding G-1
 - definition 2-33
- attributes associated with a data view 2-33
- authorities granted to the public 2-46
- authorities related to contents of a context 2-45
- authorities supported for each object 2-54
- authority
 - associated with user profiles 2-48
 - definition G-1
 - for process control instruction usage 4-8
 - in a system pointer 2-49
 - requirement for machine context 2-45
 - verification 2-58
- authorization
 - definition G-1
 - exception 2-46
 - functions 2-56
 - management 2-46
- authorization for external event handler 2-60
- authorization for external exception handler 2-60
- authorization to use the system resources 2-46
- authorizations assigned to new user profile 2-56
- authorizations associated with user profiles
 - object authorizations 2-48
 - owned objects 2-48
 - privileged instructions 2-48
 - resource authorizations 2-48
 - special authorizations 2-55
- autocall unit 6-79
- autodial unit 6-79
- automatic
 - attribute 4-1
 - definition G-1
 - ensure 7-7
 - index maintenance 5-14
 - storage 3-33
- auxiliary storage control 4-34
- auxiliary storage resource 4-33
- auxiliary storage threshold exceeded event 4-34, 4-39
- auxiliary storage, definition of G-1
- availability of exception related data 4-28
- avoiding invalid results, operand overlap 3-10

B

- back out, definition of G-1
- backward system pointer 6-15
- base, definition of G-1
- based data views 2-21, 2-34
- basic computational and branching capabilities 3-5
- basic functions for the management of data 5-2

- basic functions of modify instruction 6-32
- basic process phase 4-4
- before-image, definition of G-1
- bias, definition of G-2
- biased exponent, definition of G-2
- binary computation 3-14
- binary digits, definition of G-2
- binary elements 2-10
- binary floating-point number, definition of G-2
- binary floating-point value, definition of G-2
- binary floating-point values 2-13
- binary point, definition of G-2
- binary synchronous communications (see BSC)
- binding operand references to the actual objects 2-26
- binding, definition of G-2
- bit, definition of G-2
- block transfer attribute 4-39
- Boolean
 - instructions 3-27
 - instructions, definition G-2
 - operations 3-27
- boundary attribute 2-33
- branch form, computational instruction 3-13
- branch form, definition of G-2
- branch or indicator options 5-33
- branch point
 - and instruction pointers 2-18
 - definition G-2
 - exception handling 4-27
 - view 2-22
- branching instructions, definition of G-2
- branching operations 3-33
- broadcast capability, event signals 4-13
- BSC binary synchronous communications)
 - concepts 6-52
 - controller description eligibility list 6-18
 - definition G-2
 - network description candidate list 6-18
 - physical address checking 6-27
- building a data space template 5-3
- building an I/O configuration 6-69
- byte, definition of G-2

C

- CAI (compute array index) 3-33
- categories of machine checks 7-2
- categories of object authorities
 - authorized pointer 2-53
 - data related authorities
 - delete authority 2-53

- categories of object authorities (continued)
 - data related authorities (continued)
 - insert authority 2-53
 - retrieve authority 2-53
 - space authority 2-53
 - update authority 2-54
 - object control
 - destroy object 2-51
 - load object 2-51
 - modify damage state 2-51
 - suspend object 2-51
 - transfer ownership 2-51
 - object management
 - create cursor 2-52
 - data base maintenance 2-52
 - grant authority 2-52
 - initiate process 2-52
 - modify addressability 2-52
 - modify attributes 2-52
 - rename 2-52
 - retract authority 2-52
- categories of objects 2-1
- categories of service requests to the MSCP 6-48
- CD (controller description)
 - active count 6-66
 - contact event 6-18, 6-79
 - definition G-2
 - description 2-2
 - event 6-23
 - manual intervention event 6-79
 - object 6-23, 6-66
 - session count 6-66
- change offset value in space pointer 2-18
- changed in-use data spaces 7-6
- changes to data space entries 5-9
- changing elements within source/sink objects 6-26
- changing the number of arguments to be passed 2-40
- character string 3-25
- characteristics of a network port 6-1
- characteristics of controllers 6-1
- characteristics of space object 5-35
- cipher instructions 3-33
- class MPL limit 4-35
- classes and recipients of
 - authorizations 2-49
- classes of exceptions 4-23
- cleanup procedure, load/dump 6-62
- coexistence of locks 4-42
- coincident operand overlap 3-8
- coincident overlap
 - identical operands 3-8
 - nonidentical operands 3-8
- command specific field 6-58
- commit
 - decommit operation 5-27
 - description 5-26
 - flow chart 5-28

- commit (continued)
 - instructions 5-25
 - management 5-25
 - object 5-25
 - operation 5-26
- commit block
 - category 2-1
 - definition G-2
 - description 2-2
- commit cycle, definition of G-2
- commit identifier, definition of G-2
- commit management
 - commit block 5-25
 - commit cycle 5-27
 - commit description 5-26
 - commit object 5-25
 - commit operation 5-26
 - decommit operation 5-27
 - use 6-22
- commit transaction boundary, definition of G-2
- commitment control, definition of G-2
- common attributes of data objects 2-19
- common attributes of system objects
 - access group membership 2-8
 - context addressability 2-8
 - existence attribute 2-8
 - performance class 2-8
 - space attribute 2-8
 - system object identification 2-8
- common elements in LUD, CD, ND 6-20
- communications
 - BSC 6-8, 6-52
 - DHCF 6-13
 - SNA
 - diagrams 6-6, 6-11
 - overview 6-44, 6-48
 - X.25 6-8, 6-54
- communications device, definition of G-2
- communications error recovery 6-43
- communications system, definition of G-3
- compare length field 6-58
- compare value qualifier 4-16
- compare value, exception
 - description 2-23, 4-31
- comparison instructions 3-28
 - definition G-3
 - list 3-28
- comparison operations 3-28
- components of a program template 3-4
- composite key, data base 5-5, 5-15
- compound ODT reference, definition of G-3
- compound operand reference 3-7
- compound substring operand form 3-7
- compound substring operands 3-25
- compress access group space 4-38
- computation, floating-point 3-15
- computational
 - characteristics 3-5
 - instructions 3-5
- computational (continued)
 - operands 3-5
- computational and branching
 - capabilities 3-5
- compute array index (CAI) 3-33
- concepts, BSC 6-52
- conditional branching 3-34
- conditions for signaling an event
 - monitor 4-19
- conditions that signal the invocation
 - reference event 7-5
- configuration
 - changes 6-69
 - hierarchy rules 6-28
 - information 6-17
 - invalid exception 6-28
 - record 6-17
- configurations and states of source/sink
 - objects 6-14
- configurations defined, source/sink 6-15
- configurations of source/sink
 - objects 6-15
- conflicting locks 4-43
- connection point manager, definition of G-3
- considerations for the request I/O response
 - queue 6-42
- constant data objects 2-22
- constant data, definition of G-3
- constrained array 2-37
- constrained substring 2-38
- constrained, definition of G-3
- contents of
 - an independent index entry 5-30
 - exception description 2-23
 - journal entry 2-4, 5-23
 - machine configuration record 6-18
 - message prefix 5-33
 - message text 5-33
 - operand list 2-23
 - request I/O timestamp 6-39
 - source/sink components 6-39
 - space 2-32
 - space object 5-35
 - SSR 6-39
 - symbolic address 2-30
- context
 - addressability 2-8
 - addressed in the name resolution
 - list 2-31
 - addressing 2-31
 - authorizations 2-45
 - creation 2-44
 - definition G-3
 - description 2-2
 - destruction 2-44
 - management 2-44
 - management functions 2-44
 - qualifier in the symbolic address 2-31
- contiguous arrays 2-20

- contiguous return bit 5-11
- control and monitoring functions 4-35
- control function facilities 4-32
- control state changes 6-23
- control states, source/sink 6-22, 6-22
- controller active--LUDs in session state 6-23
- controller active--varied on LUDs state 6-23
- controller description (see CD)
- controller description, definition of G-3
- controls provided by authorization management 2-46
- conversion instructions 3-30
- conversion operations 3-29
- convert BSC to character 3-30
- convert character to BSC 3-30
- convert character to MRJE 3-31
- convert character to SNA 3-31
- convert MRJE to character 3-31
- convert SNA to character 3-31
- convert, definition of G-3
- copying date space entries 5-6
- correct execution sequence of individual instructions 4-34
- create and load command
 - object ID field, load/dump 6-59
 - reset instruction 6-61
 - use of 6-55
- create and load, load/dump 6-59
- create/destroy instructions--hierachy rules 6-27
- create process control space 4-6
- creating
 - a cursor 5-3
 - a data space 5-3
 - a data space index 5-5
 - a dump space 5-38
 - a journal 5-21
 - a program 2-9
 - a queue 5-32
 - source/sink objects in
 - sequence 6-28, 6-71, 6-72
 - source/sink objects out of
 - sequence 6-73, 6-74
 - user-defined contexts 2-44
- credited auxiliary storage, resource authorization 2-61
- cumulative authority 2-49
- current instruction in an invocation 4-29
- current MPL (eligible) 4-2
- current multiprogramming level (MPL) 4-2
- current process attributes 4-5
- current process user profile 4-5
- cursor
 - activated state 5-3
 - creating 5-3
 - de-activated state 5-3
 - definition G-3
 - description 2-2

- cursor (continued)
 - system object 2-1
- cursor, definition of G-3

D

- damage set by source/sink
 - instructions 6-66
- damage to system objects 7-8
- damaged contexts 7-7
- damaged objects 6-66
- data
 - descriptions 3-6
 - element views 2-20
 - functions 5-1
 - objects 2-10
- data attributes of the instruction
 - operands 3-5
- data base
 - definition G-3
 - journal management 5-21
 - maintenance functions 5-12
 - management 5-2
 - modifying object attributes 5-9
 - objectives 5-1
 - objects 5-1
 - performance considerations 5-10
 - recovery capabilities 7-7
 - recovery considerations 5-9
 - using functions 5-3
- data link control 6-44
- data object address resolution 2-35, 2-36
- data object addressing 2-32
- data object location attribute 2-32
- data option, exception-related 4-31
- data pointer
 - address resolution 2-35, 4-6
 - contents 2-19
 - definition G-4
 - use of 2-29
- data pointers resolved to an external scalar data object 2-35
- data-related authorities 2-50
- data space
 - category 2-1
 - definition G-3
 - description 2-2
 - function 5-12
 - organization 5-14
 - reorganized 5-10
 - template 5-3
 - used by data base 5-1
- data space entry locks 5-10
- data space entry, definition of G-3
- data space index
 - addressing 5-15
 - definition G-3
 - description 2-3

- data space index (continued)
 - entry selection routine 5-17
 - function 5-14
 - functions 5-14
 - keys 5-11, 5-15
 - maintenance 5-9
 - ordering example 5-17
 - recovery considerations 5-9
 - used by data base 5-3
- data space indexes being rebuilt 5-10
- data spaces being journaled 5-10
- data spaces indexes marked unique 5-20
- data transfer 4-33
- data types
 - binary 2-10
 - computation 3-14
 - floating point (see floating point)
 - packed decimal 2-20
 - zoned decimal 2-10
- data view declared as an element 2-20
- de-activate cursor 5-8
- de-activate logical unit control message (DACTLU) 6-50
- de-activated state of cursor 5-3
- de-activating
 - a cursor 5-5
 - a session 6-34
- deadlock
 - cause 4-45
 - definition G-4
 - detection and resolution 4-47
 - due to sequence of lock application 4-46
 - examples 4-46, 4-47
 - prevention 4-46
 - prevention rule 5-8
- default initial value 3-38
- default rounding mode, definition of G-3
- default time-out interval 4-40
- default time-out value 6-37
- default wait time-out interval
 - attribute 4-10
- deferred exception handling 4-27
- defined data views 2-21, 2-34
- definition of argument 2-39
- definition of exceptions 4-23
- definition of object 2-1
- definition of parameter 2-39
- delayed maintenance option, data space index 5-11
- deleting an entry 5-5
- denormalized number 2-14
- denormalized number, definition of G-4
- dequeue of messages from the request I/O response queue 6-41
- dequeue sequence 5-32
- dequeue wait state 5-31
- dequeue, definition of G-4
- dequeuing the message 5-31
- describing a data space 5-12
- descriptions of system objects 2-1
- destination, definition of G-4
- destroy instructions 6-27
- destroy process control space 4-6
- destroying
 - a cursor 5-5
 - a data space 5-5
 - a data space index 5-6
 - a dump space 5-6
 - a queue 5-33
 - a user profile 2-56
 - exception related data 4-28
- detached, definition of G-4
- determining the state of a source/sink object 6-34
- device controller 6-1
- device-dependent status field 6-42
- DHCF (see distributed host command facility)
- diagnostic active indicator 6-66
- diagnostic and service functions 7-3
- diagnostic state, source/sink 6-22, 6-23
- dial in
 - devices 6-79
 - operations 6-18
- dial out
 - operations 6-18
 - procedure 6-79
- dial pending state 6-24
- dialing out state 6-23
- dictionary entries 1-5
- differing-length composite keys 5-14
- direct data views 2-33
- direct data views (static and automatic) 2-21
- direct map 5-11
- direct on automatic 2-21
- direct on static 2-21
- disabled exception description 4-25
- disadvantages of authority in pointer 2-49
- dispatching processes 4-43
- display station pass-through 6-46
- dispositions of request I/O operations 6-81
- distributed host command facility (DHCF) 6-13
- dividing available resource into pools 4-33
- domain of a process 4-2
- dual role of state change/status field, source/sink 6-20
- dump attribute 2-55
- dump command 6-59
- dump restricted 2-55
- dump space
 - data 5-37
 - explanation 2-3, 5-37
 - function 5-37
 - management 5-37

dump unrestricted 2-55
dump, definition of G-4
duplicate data space index keys 5-20
duplicate key rules 5-16
duplicate physical addresses 6-27
dynamic recovery capability 7-2

E

editing instructions 3-35
editing instructions, definition of G-4
editing operations 3-34
efficient transfer of data between levels
of storage 4-33
eight-byte format, definition of G-4
element and array attributes 2-19
elements contained in a LUD 6-18
elements passed as arguments 2-39
enabled/disabled state 4-17
encapsulated program 1-2
encapsulation, definition of G-4
enqueue
definition G-4
sequence 5-32
enqueueing a message
function 5-33
moving messages 5-34
use of 5-31
enrolling users 2-56
ensure
changes to data base 5-8
data space entries 7-6
definition G-4
multiple entries 5-8
multiple inserts 5-8
object 5-9, 7-7
one entry at a time 5-8
entry in data space 5-1
entry in journal space 5-23
entry point 2-22
entry point, definition of G-4
entry, definition of G-4
error recovery, communications 6-43
error situations that cause feedback
records 6-39
error summary field 6-42
establish a switched connection 6-79
establishing a process 4-1
establishing addressability to system
object 2-30
event class 4-15, 4-16
event handler specification 4-17
event handler, definition of G-4
event handling 4-19
event identification 4-15, 4-16
event management 4-15
event monitor priority 4-17
event monitor, definition of G-4

event monitoring 4-15, 4-16
event-related data 4-21, 6-79
event rules 4-22
event signaling 4-18
event subtype 4-15, 4-16
event type 4-15, 4-16
event, definition of G-4
events 4-15, 6-27
examples
data space and associated
attributes 5-13
data space index ordering 5-17
instruction and ODT 3-3
instruction stream 3-1
journal entry 2-3, 5-23
ODT 3-2
shared data space 5-7
state change/status field 6-21
state change transition rules 6-34
exception compare value 4-26
exception description 2-23
attributes 4-24
definition G-4
examples 4-23
exception detection and signaling 4-25
exception errors 6-58
exception handler specification 4-24
exception handling 4-26
exception handling actions 4-25
exception identification 2-23, 4-31
exception management 4-23
exception occurrence flag 2-23
exception-related data 4-31
exception-specific data 4-31
exception, definition of G-4
exceptions 4-14, 6-61
exceptions and events, source/sink 6-43
excessive time required to complete I/O
instruction 6-33
exchange ID (XID) protocol 6-79
exchange identification, definition
of G-4
exclusive-allow read lock 5-7
exclusive locks on source/sink
objects 6-26
execute-only authorization 2-53
existence attribute 2-8
existence, definition of G-4
explicit address resolution 2-31
explicit request I/O instruction to the
MSCP 6-48
exponent range, definition of G-4
exponent, definition of G-4
extendable queue 5-32
extension value 5-32
external data object resolution 4-6
external data objects 2-35
external entry point, definition of G-5
external exception handling 4-28
external existence state 4-2

external scalar data views 2-19
external scalar objects 2-35
external, definition of G-4

F

FBR (feedback record)
 contents 6-42
 definition G-5
 format and contents 6-42
 load/dump 6-62
 special considerations 6-42
 use 6-40
feedback record (see FBR)
feedback record, definition of G-5
field, definition of G-5
FIFO (first in, first out) 5-16
FIFO (first in, first out) queue 5-8
FIFO ordering 5-20
file positioning 7-7
finding an entry with a data space
 index 5-6
finding an entry without a data space
 index 5-4
first pointer 6-39
first-to-last sequence 6-79
fixed-length argument list 2-23, 2-40
fixed size queue 5-32
floating point
 characteristics 2-14
 computation 3-15
 conversions 3-18
 elements 2-11
 exception conditions 3-21
 inexact result 3-23
 invalid operand 3-24
 long format 2-14
 operands 3-15
 overflow 3-21
 rounding 3-17
 short format 2-14
 underflow 3-22
 values 2-13, 2-16
 zero divide 3-23
floating-point elements
 contents 2-11
 exponent field 2-14
 format 2-14
 fraction field 2-14
 illustration 2-12
 sign field 2-14
floating-point format, definition of G-5
floating-point short format 2-14
flushing unprocessed REQIO
 instructions 6-61
forced ordering of keys between data
 spaces 5-16

forced write option 5-8, 7-6
fork character 5-5, 5-16
fork character, definition of G-5
format and contents of the feedback
 record 6-42
format of the RD (request
 descriptor) 6-57
format of zoned decimal elements 2-10
format's maximum, definition of G-5
format's minimum, definition of G-5
forms of addressing
 argument addressing 2-39
 ODT addressing 2-26
 parameter addressing 2-39
 pointer addressing
 data pointer 2-29
 instruction pointer 2-29
 space pointer 2-27
 system pointer 2-29
 process addressing 2-43
 space addressing 2-32
 system object addressing 2-30
forward and backward object pointers 6-20
forward and backward system pointers 6-15
forward recovery, definition of G-5
forward system pointer 6-15
four-byte format, definition of G-5
fraction, definition of G-5
function authorizations 2-55
function check, definition of G-5
function field 6-39
function management layer (PGM) 6-44
function of arguments and parameters 2-39
functional location of data object 2-33
functional part of object 1-2
functions of data space index 5-13
functions of request I/O 6-37
functions used to verify proper
 authorization 2-61

G

generic computational instructions 1-5
generic computational operations 3-6
generic instruction, definition of G-5
generic key addressing 5-15
generic operations attributes 3-6
generic search arguments 5-29
grant-like authority
 instruction 2-48, 2-57
group-by operations 5-4
group of elements 6-28
group of elements materialized on one
 instruction 6-28

H

- hard (unrecoverable) machine checks 7-2
- hexadecimal, definition of G-5
- hierarchy of object authorization 2-50
- high-level machine interface 1-1
- high-level operations 1-1
- homogeneous data space entries 5-1

I

- I/O instruction requires excessive time 6-33
- I/O manager (see IOM)
- I/O manager, definition of G-5
- I/O port, definition of G-5
- identification of processes 2-43
- IDL (instruction definition list)
 - definition G-6
 - used as an operand 2-23
 - used in a compound operand of a branch operand 2-43
- ignored exceptions 4-26
- immediate branch targets 2-43
- immediate data operand, definition of G-5
- immediate data value 3-5
- immediate exception handling 4-27
- immediate instruction numbers 2-43
- IMPL (initial machine program load)
 - function 7-1
- implicit address resolution 2-31
- implicit leading bit, definition of G-5
- implicit locks 4-44, 5-10
- implicit object authorizations 2-55
- implicit requests to support source/sink instructions 6-48
- implied reference to arguments 2-39
- in- and out-mapping specification 5-3
- inactive session states 6-22
- increment maximum number of entries 5-12
- increment maximum number of entries option 5-14
- independent index, definition of G-5
- independent process, definition of G-5
- index addressability to subsets of data space entries 5-17
- index management 5-29
- index system object 2-3
- indexing to subsets of entries within data spaces 5-14
- indicator form, computational instruction 3-13
- indicator form, definition of G-5
- ineligible threshold 4-37
- ineligible wait 4-2
- ineligible wait (not in current MPL) 4-2
- inexact result, definition of G-5
- inexact result, floating point 3-23
- infinity arithmetic, definition of G-6
- infinity, definition of G-6
- infinity, floating-point 2-14
- inherent machine observation functions 7-4
- inherent machine observation instructions 7-4
- initial value
 - for a pointer 2-21
 - journal sequence number 5-23
 - of space entry 2-8
- initial values 2-19
- initiate process 4-8
- initiating supervisory service requests 6-48
- initiation phase 4-2
- initiation phase option attribute 4-9
- initiation phase program 4-10
- input/output network components
 - binary synchronous communications attachments 6-8
 - local device 6-3
 - local subsystem devices 6-4
 - Multi-leaving Telecommunications Access Method
 - Multipoint Tributary 6-9
 - remotely attached devices 6-5
 - Support for MRJE 6-10
 - system to system attachment (SNA) 6-6
- insert default entries 5-12
- insert deleted entries 5-12
- insert sequentiality 7-6
- inserting an entry 5-4
- inserting dump space data 5-28
- inserting entries in independent indexes 5-30
- inserting index entries 5-30
- instruction addressing
 - instruction definition list (IDL) 2-43
 - instruction numbers 2-43
 - instruction pointers 2-43
- instruction definition list (see IDL)
- instruction number 2-18
- instruction operation code 3-1
- instruction pointer 2-18
- instruction stream 3-1
- instruction stream, definition of G-6
- instruction tracing 7-5
- instruction wait 4-2
- instruction wait access state control attribute 4-9
- instructions that address space objects 5-36
- instructions that manipulate a space pointer 2-27
- integrity and authorization 1-4
- integrity of source/sink objects 6-26
- integrity of source/sink operations 6-26

- integrity, definition of G-6
- interinvocation communications 3-49
- intermediate denormalized floating-point number, definition G-6
- intermediate result, definition of G-6
- internal exception handling 4-28
- internal form 5-15
- internal machine configuration 6-17
- internal processing phases 4-4
- interprocess communication
 - arguments and parameters 3-53
 - during events 4-15
 - when effective 4-12
- intrainvocation communications 3-53
- introduction 1-1
- invalid data space indexes 5-9
- invalid lock state exception 4-43
- invalid operand, floating-point 3-24
- invalid pointer detected by a modify operation 6-67
- invalid results, operand overlap 3-11
- invalid switched forward/backward pointers 6-67
- invalidate data space index 5-12
- invalidating a pointer 2-18
- invalidating the pointer 2-6
- invalidation (recovery capability) 7-6
- invocation
 - addressing form 02-02
 - creation 3-38
 - definition G-6
 - destruction 3-42
 - example 3-45
 - exit programs 3-44
 - reference event 7-5
 - tracing 7-5
- invocation functions 3-38
- invoking the program 3-35
- IOM (I/O manager) malfunction 6-67
- IOM malfunction is detected 6-67
- IPL cleanup 6-66

J

- journal
 - applying changes 5-24
 - entry 2-3, 5-23
 - management 5-21
 - object recovery 5-24
 - objects 5-21
 - port 2-3, 5-21
 - space 2-4, 5-22
 - status during IMPL 5-24
- journal entries 5-23
- journal entry-specific data, definition of G-6
- journal entry, definition of G-6
- journal port, definition of G-6

- journal space, definition of G-6
- journal, definition of G-6
- journalized data spaces 5-11

K

- key
 - addressing 5-15
 - constructs 4-1
 - field modification attributes 5-15
 - function of SNA 6-43
 - length 5-32
- keyed message queue 6-42
- kinds of contexts
 - machine context 2-44
 - user-defined context 2-44

L

- late bound views of data 5-1
- LD (load) commands
 - create and load command 6-55
 - dump command 6-55
 - load command 6-55
 - read object ID command 6-55
 - set context command 6-55
 - set journal command 6-55
 - set journal data command 6-55
 - set load/dump parameters command 6-56
 - set user profile command 6-55
- LD commands
 - command-specific field 6-58
 - error processing 6-60
 - object ID field 6-59
- LEAR (lock exclusive allow read)
 - length of a parameter list 2-41
 - length of binary elements 2-10
 - length of exception compare value 4-26
 - length of RD 6-57
 - length of the key 6-42
 - length suboperand 3-7
- LENR (lock exclusive no read) 4-41
- levels of storage 4-33
- LIFO (last in, first out) 5-16
- LIFO ordering 5-20
- limit of temporary storage allocated by a process 4-34
- limiting the number of concurrently executing processes 4-35
- limiting the number of processes 4-33
- limiting the total space allocated to objects 4-39
- linkage of one object type to another object type 6-14
- list of return targets 2-43
- listener of the event 6-79

- load attribute 2-55
- load command 6-59
- load command, cleanup procedure 6-62
- load/dump (LD)
 - authority 6-62
 - commands 6-55
 - compress bit, object control field 6-58
 - considerations 6-54
 - context bit, object control field 6-58
 - data base networks 6-62
 - function 6-54
 - functions 5-38
 - journal bit, object control field 6-58
 - journal entries 6-64
 - journal spaces 6-64
 - load/dump networks 6-62
 - networks 6-62
 - performance 6-64
 - recovery 7-7
 - session types 6-56
- local work station, definition of G-6
- locating an exception description 4-25
- location of a constant data object 2-22
- location where control is passed when exception occurs 2-23
- lock
 - a data space to a process 5-7
 - a space location 4-44
 - allocation rules 4-41
 - coexistence graph 4-42
 - definition G-7
 - exclusion graph 4-43
 - exclusive allow read (LEAR) 4-41
 - exclusive no read (LENR) 4-41
 - materialize 4-44
 - request granting algorithm 4-41
 - shared read (LSRD) 4-41
 - shared read only (LSRO) 4-41
 - shared update (LSUP) 4-41
 - states for a data space 5-7
- lock a space location 4-44
- locking instructions 4-34
- locking multiple entries 5-8
- locking objects to processes 4-34
- logical character operations 3-35
- logical unit description (see LUD)
- logically reorder data space entries 5-14
- long format values 2-16
- long format, definition of G-7
- LSRD (lock shared read) 4-41
- LSRO (lock shared read only) 4-41
- LSUP (lock shared update) 4-41
- LU (logical unit) 6-1
- LUD (logical unit description)
 - definition G-7
 - object 6-66
 - session state changes 6-33
 - states 6-22
- LUD, CD, and ND identification 6-20
- LUD, CD, ND type 6-20

M

- machine attribute modification
 - authorizations 2-55
- machine attribute, definition of G-7
- machine attributes 7-1
- machine check
 - definition G-7
 - event 7-3
 - event-related data 7-3
 - exception 7-3
 - function(MCF) 7-2
- machine checks 7-2
- machine configuration record 6-16
- machine context 2-2
- machine context, definition of G-7
- machine default position 2-33
- machine-dependent data 4-31
- machine event signaling 4-18
- machine exception identification 4-24
- machine interface, definition of G-7
- machine main storage pool 4-37
- machine observation functions 6-17, 7-4
- machine-related events 4-15
- machine resources 1-1
- machine services control point (see MSCP)
- machine services control point, definition of G-7
- machine-signaled exceptions 4-25
- machine storage 4-33
- machine support functions 7-1
- machine termination, definition of G-7
- machine-to-programming transition 7-1
- machine-wide and class MPL controls 4-35
- machine-wide MPL limit 4-35
- machine-wide signal domain indicator 4-19
- magnitude, definition of G-7
- major data object attributes
 - data object mapping attribute 2-21
 - element and array attributes 2-20
 - external attribute 2-22
 - initial values 2-21
- major functional layers defined by SNA 6-43
- major parts of a program 2-9
- managing dump spaces 5-38
- managing the network and device facilities 6-26
- manual answer state 6-24
- manual dial start state 6-24
- manual start data state 6-25
- mapping
 - attribute 2-19
 - attributes 2-21
 - definition G-7
 - template 5-3
- materialization of data base object
 - attributes 5-9

- materialization of data base object
 - statistics 5-8
- materialize
 - definition G-7
 - instructions 6-28, 7-5
 - journal entries 5-23
 - LUD template 6-29
 - machine configuration record 6-17
 - option value assigned to each element 6-28
 - process attributes 4-7
 - source/sink objects 6-28
- materialize locks 4-43
- materialize queue messages 5-34
- materialize, definition of G-7
- materializing authority 2-58
- materializing contexts 2-45
- materializing queue attribute 5-32
- materializing space data 5-38
- maximum
 - length of
 - a space 5-35
 - an index entry 5-30
 - number of
 - locks 4-42
 - messages 5-32
 - processes, assigned to a class 4-35
 - signals to be retained 4-17, 4-18
 - size of
 - a queue 5-32
 - a queue message 5-33
 - exception-related data 4-31
 - messages 5-32
 - temporary auxiliary storage
 - allowed 4-40
 - temporary auxiliary storage allowed
 - attribute 4-10
 - time-out value 6-37
- maximum process time allowed
 - attribute 4-10
- MCF (machine check function) 7-2
- membership in access group 2-8
- message content indicator 5-32
- message key 5-33
- message prefix 5-33
- message text 5-33
- message, definition of G-7
- messages on the request I/O response
 - queue 6-40
- messages, source/sink 6-42
- method of recovery 6-83
- method used to recover from the terminating
 - error condition 6-83
- minimum authority requirement 2-32
- minimum time-out value 6-37
- MODCD (dial out) command 6-24
- modes of addressing 1-5
- modifiable element 6-30
- modification control indicators 4-10
- modification sequences 6-32
- modify CD 6-67
- modify instruction functions 6-32
- modify instruction issued after
 - error 6-83
- modify instructions 6-30
- modify LUD 6-67
- modify LUD sessions for LD 6-61
- modify LUD template 6-31
- modify ND 6-67
- modify process attributes 4-8, 4-14
- modify source/sink objects 6-30
- modify time-out values 6-37
- modifying an instruction pointer 2-29
- modifying authorization 2-56
- modifying data base object attributes 5-9
- modifying multiple elements 6-30
- modifying pointers 2-26
- modifying space data 5-37
- modifying the symbolic address of an
 - object 2-30
- monitor domain 4-17
- monitoring events for system objects 4-16
- monitoring exceptions 4-23
- monitoring functions facilities 4-32
- monitoring MPL Activity 4-37
- move messages between queues and
 - processes 5-34
- movement instructions 3-29
- moving data units between origins and
 - destinations 6-44
- moving messages
 - dequeuing a message 5-34
 - enqueueing a message 5-34
- MPL
 - classes 4-35
 - process states 4-2
 - rules 4-36
- MSCP (machine services control point)
 - definition G-7, 6-48
 - peer station 6-51
 - primary 6-49
 - secondary 6-50
 - role 6-50
 - sessions 6-50
- multiple argument list entries 2-40
- multiple locked entries 5-8
- multiple processes lock request granting
 - algorithm 4-43
- multiple state transitions 6-34
- multiple views of data 5-1
- multiprogramming level control 4-33, 4-35
- multiprogramming support 1-5

N

- name resolution function 2-2
- name resolution list (NRL) 2-31
- name resolution list, definition of G-7
- name resolution, definition of G-7
- NaN, definition of G-8
- ND (network description)
 - candidate lists 6-18
 - definition G-7
 - description 2-5
 - object 6-22
 - object category 2-1
 - object type 6-2
- ND active count 6-66
- ND object 6-66
- negative infinity, definition of G-7
- network active states 6-24
- network addressable unit, definition of G-8
- network boundary bits 6-63
- network description (see ND)
- network port 6-1
- network port, definition of G-8
- network, definition of G-8
- networking bit 6-63
- no operation (NO OP) 3-35
- no operation and skip 3-35
- noncontiguous arrays 2-20
- nonoverlap instructions 3-9
- nonrecoverable error detected, load/dump 6-56
- nonsupported session state changes 6-37
- nonzero time-out value 6-37
- NOOP (No Operation) 3-35
- normal recovery from damaged objects 6-66
- normalized number 2-14
- normalized number, definition of G-8
- not-a-number, definition of G-8
- not-a-number, floating-point 2-14
- NRL (name resolution list) 2-31, 4-6
- null substring 2-38
- number of elements contained in an array 2-20
- number of event monitors attribute 4-9
- number of locks currently held by the process value 4-12
- number of RDs within the SSR 6-57
- number of signals pending value 4-19

O

- object address resolution 4-6
- object authorities
 - authorized pointer 2-48
 - delete 2-48
 - insert 2-48
 - object control 2-48
 - object management 2-48
 - retrieve 2-48
 - space 2-48
 - update 2-48
- object authority, definition of G-8
- object authorization 2-48, 2-61
- object authorization qualifier 2-31
- object authorization states 2-50
- object authorization, definition of G-9
- object concepts 1-1
- object contents 6-18
- object control
 - authorities 2-50
 - functions 2-1
 - states 6-22
- object creation and destruction 6-28
- object definition 2-1
- object definition table (see ODT)
- object destroyed exception 5-33, 5-36
- object header data 6-20
- object identification 2-8
- object isolation 7-8
- object locks 4-13
- object management authorities 2-50
- object management authority
 - authority 2-52
 - authorization 2-48
 - hierachy 2-50
 - operations 2-52
- object mapping table 3-4
- object modification limitations 6-17
- object movement 3-29
- object name 2-8
- object-oriented architecture 1-1
- object owner, definition of G-9
- object recovery list 7-7
- object-related events 4-15
- object states 6-20
- object subtype 2-8
- object subtype, load/dump 6-59
- object type 2-8
- object type, load/dump 6-59
- object types
 - controller description (CD) 6-2
 - logical unit description (LUD) 6-2
 - network description (ND) 6-2
- object usage states 6-22
- object, definition of G-8
- objects addressable by the machine context 2-44

- objects that can be processed by the LD function 6-54
- objects, transient 4-37
- observation functions 7-4
- observation of machine execution 7-4
- obtaining addressability to system objects 2-2
- occurrence of machine events 4-18
- ODT (object definition table)
 - definition of an object 2-9
 - directory vector, definition of G-9
 - extender string, definition of G-9
 - use 3-2
- ODV (ODT directory vector) 3-2
- ODV, definition of G-9
- OES (ODT entry string) 3-2
- OES, definition of G-9
- offset calculations within space 2-38
- offset to variable parameters 6-40
- oldest event 4-20
- OMT (see object mapping table)
- operand field, definition of G-9
- operand list 2-23
- operand list (if used as an argument list) 2-39
- operand list used as a parameter list 2-23
- operand of an instruction 3-1
- operand overlap
 - definition G-9
 - description 3-7
 - examples 3-10
- operands defined in the ODT 3-5
- operands in computational instructions 3-5
- operands referenced in computational instructions 3-5
- operation code extender field 3-1
- operation code extender field, definition of G-9
- operation code field, definition of G-9
- operational authorities 2-51, 2-53
- operational parameter elements 6-36
- operational parameters 6-32
- operations that cause processing to be restarted 6-83
- operations that require object control authorization 2-51
- operations that require object management authority 2-52
- operator intervention 6-82
- operator intervention required event 6-82
- optimizing data space index usage 5-11
- option value 6-16
- optional computational instruction forms 3-12
- optional function of TMPF 7-1
- optional round forms 3-12
- optional user data 4-16

- options available in positioning a cursor 5-4
- options defined by exception
 - description 4-24
- options on insert 5-30
- options on inserting entries in independent indexes 5-30
- order of processing for request descriptors 6-57
- ordered relationship among object types 6-14
- ordering attributes 5-5
- ordering duplicate keys 5-14
- ordering of keys in binary collating sequence 5-34
- orderly shutdown of systems 6-69
- ordinal entry number, definition of G-9
- ordinal number identifier 5-1
- ordinal positioning 5-14
- organization of data space 5-14
- other errors 6-61
- outstanding requests 6-83
- overflow, floating-point 3-21
- overlap instructions 3-8
- overlapped data base operations 5-10
- overview of the load/dump function 6-56
- ownership of objects 2-46

P

- packed decimal
 - computation 3-14
 - elements 2-20
 - format 2-20
 - numbers 2-20
- packet switching data network (PSDN)
 - definition 6-2
 - SNA example 6-6
 - source/sink example 6-68
 - SVC 6-81
 - switched pointers 6-18
 - X.25 example 6-55
- parameter
 - data elements 2-40
 - data views 2-21, 2-35
 - list 2-41, 3-51
 - list definition G-9, 2-41
- parameter list, definition of G-9
- parameter lists defined as internal or external 2-41
- parameters 2-39, 3-48
- partial damage
 - exception and event 6-67
 - indicator 6-66
 - recovery 6-67
- partial overlap 3-7
- partial search argument 5-29

partial system object damage
 exception 6-37
 partitioning main storage 4-37
 PASA (process automatic storage area)
 contents 3-40
 size 3-40
 updating 3-40
 use 3-38, 4-1
 passing an argument 2-39
 passing exception data 4-31
 PCO (process communication object) 4-10, 4-12
 PCS (process control space)
 category 2-1
 definition G-10
 description 2-5
 instructions 4-7
 use of 4-1
 PDEH (see process default exception handler)
 PDT (process definition template) 4-1
 peer station 6-1, 6-51
 performance class 2-8
 performance considerations, data
 base 5-10
 permanent auxiliary storage limit 4-39
 permanent object 2-8
 physical address 6-20
 physical I/O configuration changes 6-17
 pointer (PTR), definition of G-9
 pointer arrangements 6-15
 pointer data objects 2-18, 2-26
 pointer data types 2-18
 pointer types 2-26
 pointers as search arguments 5-30
 pointers contained in enqueued messages 5-32
 polling for messages on multiple queues 5-34
 position attribute 2-33
 positioning of cursors 7-7
 positioning options without data space index 5-6
 positive infinity, definition of G-9
 potential deadlock situation 5-8
 power off state 6-22
 power, definition of G-9
 preferred sequence for creating source/sink objects 6-28
 preferred sequence for destroying source/sink objects 6-28
 preventing deadlock situations 5-8
 primary SDLC station 6-51
 primary station, definition of G-9
 priority attribute 4-35
 priority attribute, optimization 2-25
 priority field in the SSR 6-39
 priority process attribute 4-33
 priority, definition of G-9
 private authority, definition of G-9
 private authorizations 2-49
 privileged instruction
 authorization 2-61
 definition G-10
 description 2-56
 exception 2-56
 problem determination 7-4
 problem phase 4-2, 4-4
 problem phase option attribute 4-9
 problem phase program 4-11
 procedure for using the switched communications network 6-81
 process access group 4-11
 process access group option attribute 4-9
 process adopted user profile 2-59
 process attributes 4-9, 4-40
 process authorization 4-5
 process automatic storage area (see PASA)
 process auxiliary storage limit process attribute 4-39
 process communication object (PCO) 4-10, 4-12
 process control attribute 2-55
 process control attributes 4-9
 process control instruction characteristics 4-14
 process control instructions 4-7
 process control space (see PCS)
 process default exception handler (PDEH) 4-11
 process default exception handler option attribute 4-9
 process definition template (PDT) 4-1
 process domain 4-2
 process events 4-16
 process exception handling 4-14
 process external existence state 4-40
 process initiation steps 4-2
 process interrupt pending status 4-11
 process interruption facility 4-13
 process isolation 7-8
 process management 4-1
 process management instructions 4-6
 process multiprogramming level class ID attribute 4-10
 process name resolution list (NRL) 4-10
 process name resolution list option attribute 4-9
 process performance attributes 4-12
 process phases 4-4
 process pointer attributes 4-10
 process priority attribute 4-10
 process-related instructions 4-14
 process resource usage attributes 4-12
 process states 4-2
 process static storage area (see PSSA)
 process static storage option attribute 4-9
 process status indicators 4-11

- process storage pool identification
 - attribute 4-10
- process structure 4-1
- process termination status 4-11
- process type attribute 4-9
- process user profile
 - authority verification 2-59
 - general information 2-47
 - pointer attribute 4-10
 - process authorization 4-5
- process, addressing form 2-26
- process, definition of G-10
- processing an MODLUD (reset) instruction,
 - load/dump 6-62
- processing mode 5-10
- processing order of request
 - descriptors 6-57
- processing when entering or leaving the
 - active state 4-36
- processor eligibility definition class,
 - definition of G-10
- processor resource 4-33
- program
 - activation 3-36
 - category 2-1
 - creation 3-1
 - definition G-10
 - description 2-5
 - destruction 3-4
 - execution 3-35
 - functions 3-1
 - management 3-1
 - materialization 3-4
 - object descriptions 2-9
 - objects 2-9
 - observability deletion 3-4
 - optimization 3-3
 - qualifier 2-36
 - template 3-1
 - variable attributes 1-1
- program adopted user profile 2-59
- program defined initial value 3-38
- programming conventions for reducing
 - deadlocks 4-46
- progression through states 6-22
- propagated adopted user profile 2-61
- propagated user profile, definition
 - of G-10
- propagation of adopted user profile
 - authorities 2-47
- properly terminate the appropriate
 - elements 6-69
- PSDN (see packet switching data network)
- PSSA (process static storage area)
 - contents 3-37
 - extending 3-37
 - locating 3-37
 - use 3-35, 4-1
- PTR (pointer) G-9
- public authority 2-46, 2-59

- public authority, definition of G-10
- purged access group 4-36
- purpose of object authorities 2-50

Q

- qualified symbolic address 2-30
- qualifying events 4-16
- qualifying exception description 4-25
- queue
 - category 2-1
 - definition G-10
 - description 2-5
 - extended event 5-32
 - functions 5-33
 - instructions 5-32
 - management 5-31
 - message limit exceeded exception 5-32
 - overflow action indicator 5-32
 - type indicator 5-32
 - use of 4-12
- queue, definition of G-10
- queuing functions
 - dequeuing messages 5-33
 - enqueueing messages 5-33
- quiesce, definition of G-10
- quiesced session state 6-22
- quiescing a session 6-33

R

- RD (request descriptor)
 - count 6-40
 - number 6-42
 - number for exception field 6-58
- reactivating a quiesced session 6-33
- read object ID command 6-59
- read object ID, load/dump 6-59
- reason for locking objects to
 - processes 4-34
- rebuild data space index 5-12
- receiver operand, definition of G-10
- reclaim lost objects 7-8
- recommended sequence for creating
 - source/sink objects 6-28
- recommended sequence for destroying
 - source/sink objects 6-28
- record, definition of G-10
- recoverable error 6-61
- recoverable error processing,
 - load/dump 6-56
- recovery capabilities
 - data base 7-7
 - system 7-7
- recovery from partial damage 6-67
- recovery functions 7-7

- recovery method 6-83
- recovery/resource activation state
 - CD 6-24
 - indication 6-22
 - LUD 6-23
 - ND 6-25
- recovery, definition of G-10
- reducing deadlocks 4-45
- referencing parameter objects 2-39
- referring to a system object 2-7
- relationships between argument-parameter binding 2-41
- relative instruction number 2-18, 2-43
- relative positioning 5-14
- relative positioning, definition of G-10
- remote work station, definition of G-10
- remove, definition of G-10
- removing a queue 5-33
- REQRQ (request I/O response queue) 6-40
- request control field 6-39
- request descriptor (see RD)
- request descriptor count 6-40
- request I/O
 - continue instruction 6-39
 - error recovery examples 6-82
 - function 6-37
 - instruction 6-37
 - response queue 6-38
 - response queue and feedback record 6-40
 - syntax 6-38
- request I/O and request I/O response queue relationship 6-41
- request I/O variable parameter 6-40
- request ID 6-39, 6-42
- request information unit (see RIU)
- request key 6-40
- request path operation 6-26
- request priority 6-39
- requests or responses to the MSCP 6-49
- requirement for operational authorities 2-53
- reserved values, definition of G-10
- reserved, definition of G-10
- reset
 - access group 4-38
 - data space 5-12
 - partial object damage 6-67
- reset session
 - command 6-34
 - definition G-10
 - process 6-33
 - state 6-22
- resigned exception 4-25
- resolved pointers 2-26
- resource authorization 2-56, 2-61
- resource management 4-32
- resource management attributes 4-10
- resource management control functions 4-32

- resource management monitor functions 4-32
- resources associated with user profiles 2-48
- resources managed by the control and monitoring functions 4-33
- resources, control and monitoring 4-33
- response queue, definition of G-10
- responsibilities of the user of source/sink objects 6-68
- restrictions for request I/O response queue 6-42
- result field, definition of G-10
- results of ignored exceptions 4-26
- resume process 4-8, 4-14
- retracting the authorizations held by user profile 2-57
- retrieve exception related data 4-28
- retrieving an entry 5-1, 5-4
- retrieving dump space data 5-38
- retrieving multiple entries 5-4
- return from exception action code 4-29
- return list 2-43
- return target 2-43
- returns from exception handling 4-29
- rights to an object 2-46
- RIU (request information unit)
 - segment count 6-42
 - use 6-40
- round form, computational instruction 3-12
- round to nearest, definition of G-11
- round toward negative infinity, definition of G-11
- round toward positive infinity, definition of G-11
- round toward zero, definition of G-11
- rounding mode, definition of G-11
- rounding, definition of G-11
- routing data units between origins and destinations 6-44
- runaway situations 5-14

S

- scalar data objects 2-10
- scalar objects located in static spaces 2-35
- scalar, definition of G-11
- scan instructions 3-28
- search argument size 5-29
- searching for index entries 5-29
- second pointer, source/sink 6-39
- secondary requirement for configuration changes 6-28
- secondary SDLC station 6-51
- secondary station, definition of G-11
- select/omit, definition of G-11

- selection criteria, data space
 - entries 5-17
- selection routine for data space index
 - entries 5-17
- sequence for performing a modify
 - instruction 6-32
- sequence of load/dump operation 6-56
- sequence of states maintained for the
 - CD 6-23
- sequence of states maintained for the LUD
 - object 6-22
- sequence to vary off source/sink
 - objects 6-25
- sequence to vary on source/sink
 - objects 6-25
- sequence used to perform work on an I/O
 - device 6-38
- sequencing through process phases 4-5
- sequential addressing 5-15
- sequential list of ports (NDs) 6-18
- sequentiality of updates 7-6
- service attribute 2-55
- service requests to the MSCP 6-48
- session is changed from load to dump or
 - dump to load 6-61
- session state changes 6-81
- session states in the LUD objects 6-33
- session, definition of G-11
- set access state 4-38
- set context 6-58
- set context command 6-59
- set journal data, load/dump 6-59
- set journal, load/dump 6-59
- set user profile 6-58
- set user profile command 6-59
- severe errors 6-60
- severely damaged or destroyed objects 7-7
- shared data spaces 5-7
- shared read lock 5-7
- shared usage of source/sink objects 6-68
- sharing data within a system object 4-44
- sharing system objects 4-34
- short form
 - computational instruction 3-12
 - definition G-11
 - option 4-17
- short format values 2-15
- short format, definition of G-11
- sign, definition of G-11
- signal event control mask attribute 4-9
- signal retention option 4-17
- signaled instruction number 4-31
- signaled invocation address 4-31
- signaling an event to a process 4-19
- signaling by signal event
 - instruction 4-18
- signaling instruction number 4-31
- signaling invocation address 4-31
- signed and unsigned numeric key
 - fields 5-14
- signed exponent, definition of G-11
- signed zero, definition of G-11
- significance of synchronous and
 - asynchronous instructions 6-81
- significand, definition of G-11
- simple deadlock 4-46
- simple ODT reference, definition of G-11
- single character constants 5-5
- size of message data area 6-42
- size of message text 5-33
- size of response queue 6-42
- SNA (systems network architecture)
 - concepts 6-44
 - control messages
 - MSCP to logical unit (primary) 6-50
 - MSCP to physical unit (primary) 6-50
 - SSCP to logical unit (secondary) 6-51
 - SSCP to physical unit
 - (secondary) 6-50
 - definition G-12
 - supervisory services support (MI) 6-48
 - transmission management layer 6-44
- soft (recoverable) machine checks 7-2
- source operand, definition of G-12
- source/sink
 - create instructions 6-27
 - data (SSD) 6-40
 - definition G-12
 - examples 6-68
 - functions 6-1
 - instruction usage 6-26
 - instructions 6-26
 - instructions that set object
 - damage 6-66
 - management 6-1
 - object recovery 6-66
 - object state changes 6-22
 - object subtypes 6-14
 - object types 6-26
 - objects 6-1
 - objects that support the I/O
 - configuration 6-26
 - request (SSR) 6-39
 - user responsibilities 6-68
 - source/sink data (SSD) 6-40
 - source/sink management 6-1
 - source/sink request (see SSR)
 - source/sink request, definition of G-12
 - source/sink resource, definition of G-12
 - source/sink, definition of G-12
 - sources for object authorities 2-48
 - space
 - addressing 2-26
 - authorization 2-48
 - category 2-1
 - description 2-6, 2-7
 - space addressing violation exception 2-38
 - space attribute materialization 5-36
 - space attribute modification 5-36
 - space creation 5-36

- space data 5-36
- space data modification 5-37
- space data views 5-36
- space destruction 5-36
- space extent checking 2-39
- space functions 5-35
- space management 5-35
- space object 1-3
- space object characteristics 5-35
- space pointer (SPP)
 - contained in feedback record 6-42
 - data type 2-18
 - definition G-12
 - use of 2-27
- space pointer machine object
 - description 2-24
 - optimization priority attribute 2-25
 - pointer addressing 2-27
 - verification 2-24
- space pointer, definition of G-12
- space, definition of G-12
- spaces 5-35
- special authorizations
 - assigned to user 2-55
 - associated with user profile 2-48
 - summary 2-61
- specific elements in LUD, CD, ND 6-20
- specific event monitoring 4-15
- specifying a context during object creation 2-44
- specifying contexts for system object
 - address resolution 2-31
- specifying objects to be journaled 5-22
- SPP (see space pointer)
- SSD (source/sink data)
 - contents 6-40
 - definition 6-40
- SSR (source/sink request)
 - contents
 - data 6-39
 - pointers 6-39
 - definition G-12
 - use 6-38
- standard feedback record, load/dump 6-62
- state change functions tolerant or partial damage 6-67
- state change rules 6-22
- state change/status field 6-20
- state change transition rules 6-34
- state transition diagram 4-2
- states in which optional parameters can be modified 6-32
- states of a process 4-35
- states of the CD 6-24
- states of the LUD 6-22
- states of the ND 6-24
- static storage 3-35
- static, definition of G-12
- station in a switched network 6-18
- status of process 4-11
- storage limits 4-39
- storage resource 4-33
- storage resource functions 4-37
- string, definition of G-12
- subinvocations 3-47
- subordinate processes identification 4-12
- subscript compound operand form 2-37
- substring addressing 2-37
- substring compound operand 2-37
- summary
 - event rules 4-22
 - lock granting rules 4-42
 - option attribute 4-9
 - program 4-10
 - state change transition rules (LUD) 6-35
- supervisor and control functions 4-1
- supervisory service request flow 6-49
- supplying a set of attributes to computational operands 3-7
- support for multiprogramming 1-5
- support for source/sink object types 6-26
- supported authorities for each object 2-54
- suspend object attribute 2-55
- suspend process 4-7, 4-14
- suspend restricted 2-55
- suspend unrestricted 2-55
- suspend, definition of G-12
- suspended session state 6-22
- suspended state 4-2
- suspending a session 6-33
- switched connection 6-18
- switched connection forward and backward object pointers 6-20
- switched enable state 6-23
- switched forward and backward pointers 6-18
- switched network considerations 6-17
- symbolic address 2-30
- symbolic address of an object used for context addressing 2-44
- symbolic address of context 2-31
- symbolic address of external scalar object 2-35
- symbolic address of the external data object 2-22
- symbolic name, definition of G-12
- synchronize, definition of G-12
- synchronous and a synchronous instructions 6-81
- synchronous data link control, definition of G-12
- synchronous event handling 4-20
- synchronous initiation step 4-2
- synchronous session state changes 6-81
- system default not-a-number, definition of G-12
- system integrity 1-4

- system object
 - access group 2-1
 - address resolution 2-30, 4-6
 - address resolution functions 2-30, 2-31
 - attributes 2-8, 4-39
 - characteristics 2-6
 - commit block 2-1, 2-2
 - context 2-1
 - controller description 2-1
 - cursor 2-1
 - data space 2-1
 - data space index 2-1
 - definition G-12
 - descriptions 2-1
 - dump space 2-1, 2-4
 - index 2-1
 - journal port 2-1, 2-3
 - journal space 2-1, 2-4
 - locks 4-41
 - logical unit description 2-1
 - network description 2-1
 - object authorities 2-54
 - process control space 2-1
 - program 2-1
 - queue 2-1
 - space 2-1
 - user profile 2-1
- system pointer
 - addressing 2-30
 - authority attributes 2-60
 - definition G-12
 - initial values 2-21
- system recovery capabilities 7-7
- System/38 instruction set 1-1
- System/38 support functions 7-1
- systems network architecture (see SNA)
- systems network architecture, definition of G-12

T

- tag bits 2-6
- target invocation 4-29
- target object of a request I/O operation 6-38
- template components 3-1
- template header 3-1
- template size specification 6-20
- template, definition of G-12
- temporary object 2-8
- temporary object, definition of G-12
- temporary objects 5-10
- terminate instruction 4-8, 4-14
- terminate machine processing function (TMPF) 7-1
- terminate process 4-8, 4-14
- terminating error conditions 6-83
- terminating error feedback record 6-82

- terminating error indicated in the feedback record 6-83
- termination phase 4-2
- third pointer, source/sink request 6-39
- threshold values 4-37
- time-out value 6-67
- time slice
 - attribute 4-35
 - end access state control attribute 4-9
- timer events 4-16
- TMPF (terminate machine processing function) 7-1
- TMPF optional function 7-1
- total processor time allowed 4-40
- total processor time used value 4-12
- total temporary auxiliary storage used attribute 4-12
- trace functions 7-4
- tracing instructions 7-5
- tracing invocations 7-5
- transferring locks 4-44
- transferring ownership of program 2-60
- transient attribute 4-39
- transient pool 4-37
- transmission management layer (MI) 6-44
- transmission subsystem (MI) 6-44
- treatment for previously issued request I/O instructions 6-39
- two or more entries with duplicate keys 5-16
- type of intervention required 6-82
- types of addressing for data space indexes 5-15
- types of objects addressed by machine context 2-7
- types of searches 5-29
- types of system objects that can be locked 4-41
- types of tracing 7-5
- typical contents of LUD 6-18

U

- ultimate end object 6-14
- ultimate object of I/O transactions 6-1
- unattached journal spaces, load/dump 6-65
- unbiased exponent, definition of G-12
- unconditional branching 3-33
- underflow, floating-point 3-22
- unique key rule 5-16
- unique name for user profile 2-56
- unique physical address within each object type 6-27
- unique symbolic identification 2-2
- uniqueness checking 6-27
- uniqueness of space object 5-35
- uniqueness, object type 6-27
- unit of transfer 5-10

- unit specification 5-11
- unlock a space location 4-44
- unlocking system objects 4-45
- unordered, definition of G-13
- unpredictable results in instructions 3-7
- unrecoverable machine check 7-2
- unresolved pointers 2-26
- updating a communications station 6-75, 6-76, 6-77
- updating an entry 5-5
- updating the PASA 3-40
- usage states 6-22
- use of modify instruction 6-26
- use of modify time-out value 6-37
- use of request I/O instruction 6-26
- use of source/sink objects and instructions 6-68
- use of the materialize instruction 6-26
- user data 3-3
- user data, timer events 4-16
- user-defined events 4-18
- user-defined recovery functions 7-2
- user information after IPL 5-10
- user profile 2-6
 - definition G-13
 - description 2-6, 2-46
 - event option attribute 4-9
 - interval attribute 4-10
- user-signaled exceptions 4-25
- user specified queue size 5-32
- user's interface to the load/dump function 6-56
- user's processing environment 2-46
- uses for independent indexes 5-29
- uses for resource management 4-32
- uses of system object locks 4-41
- using data base function 5-3
- using pointers as data 2-26
- using the NRL for address resolution 2-32

V

- valid sign encoding 2-11
- values materialized by Materialize Process
 - Attributes instructions 4-40
- variable addressability to instructions 2-43
- variable branch targets 2-18
- variable branching 3-34
- variable-length argument list 2-23, 2-40
- variable-length keys 5-15
- varied off/power on state 6-22
- varied off state 6-23
- varied on/no session state 6-22
- varied on state 6-24
- vary off 6-67
- vary off source/sink objects
 - sequence 6-25

- vary on all of the stations 6-78
- vary on all the communications lines 6-78
- vary on/off sequence 6-25
- vary on pending (with LUDs pending) state 6-23
- vary on pending state 6-22
- vary on source/sink objects sequence 6-25
- vary on the logical units that are eligible for use 6-78
- vector, definition of G-13
- verifying object authorities 2-60
- view 2-9
- view, definition of G-13

W

- Wait on Time instruction 4-8
- when the operator intervention required event is signaled 6-83
- work station, definition of G-13

X

- X.25 communications
 - basic concepts 6-55
 - ND states 6-26
 - source/sink management 6-1
 - SVC 6-81
- XID (exchange identification) 6-27

Z

- zero divide, floating-point 3-23
- zone or digit force 5-15
- zoned decimal
 - computation 3-14
 - elements 2-10



READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

If your comment does not need a reply (for example, pointing out a typing error) check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s):

Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or
Organization _____

Address _____

No postage necessary if mailed in the U.S.A.

City _____ State _____ Zip Code _____

Phone No. (_____) _____

Area Code _____

Fold and tape. **Please do not staple.**

Cut Along Line



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 40 / ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
Rochester, Minnesota, U.S.A. 55901

Fold and tape. **Please do not staple.**

Cut Along Line





IBM System/38 Functional Concepts Manual (File No. S38-01) Printed in U.S.A. GA21-9330-4

GA21-9330-4



GA21-9330-4