

SY21-0889-5

File No. S38-01

IBM System/38

IBM System/38

Vertical Microcode Logic Overviews and Component Descriptions Manual



SY21-0889-5

File No. S38-01

IBM System/38

IBM System/38

**Vertical Microcode Logic Overviews and
Component Descriptions Manual**

Sixth Edition (September 1985)

This is a major revision of, and makes obsolete, SY21-0889-4. X.25 data link information was added as a new chapter in Chapter 34. Existing Chapters 34 through 37 are renumbered to accommodate this addition. All other changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

This publication provides an overview of the vertical microcode components and a description of the functions within the vertical microcode components. Use this publication only for the purposes stated in *About This Manual*.

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 below. Requests for copies of IBM Publications and for technical information about the system should be made to your IBM representative or to the branch office serving your locality.

This publication could contain technical inaccuracies or typographical errors. Use the Reader's Comment Form at the back of this publication to make comments about this publication. If the form has been removed, address your comments to IBM Corporation, Information Development, Department 245, Rochester, Minnesota 55901. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

ABOUT THIS MANUAL	ix		
Purpose of This Manual	ix		
Organization of This Manual	ix		
If You Need More Information	ix		
VERTICAL MICROCODE OVERVIEW	0-1		
Data Function	0-2		
Commit Management	0-2		
Data Base Management	0-2		
Independent Index Management	0-3		
Journal Management	0-3		
Queue Management	0-3		
Space Object Management	0-3		
Internal Machine Function	0-4		
Storage Management (Auxiliary and Main)	0-4		
Machine Index Management	0-6		
Machine Support Function	0-6		
Initialization/Termination Management	0-6		
Machine Check Management	0-6		
Machine Observation Management	0-6		
Service and Installation Management	0-7		
Object Function	0-7		
Authorization Management	0-7		
Context Management	0-7		
Recovery Initialization	0-7		
Program Control Function	0-8		
Program Execution Management	0-8		
Program Management	0-8		
Source/Sink Function	0-9		
Instruction Processors	0-9		
Machine Services Control Point	0-9		
I/O Managers	0-9		
Source/Sink Data Areas	0-10		
Supervisor and Control Function	0-11		
Event Management	0-11		
Exception Management	0-11		
Process Management	0-11		
Resource Management	0-12		
Common Function	0-13		
RELATIONSHIP OF COMPONENTS	0-14		
Alternate IMPL	0-14		
Internal Microprogram Load	0-14		
Vertical Microcode and the System/38 Instruction Set	0-15		
DATA FUNCTION	1-1		
COMMIT MANAGEMENT	1-1		
Introduction	1-1		
Transactions Under Commitment Control	1-1		
IMPL Recovery	1-2		
System/38 Instruction Support	1-3		
Journal Support	1-4		
Cursor Support	1-5		
Data Space Support	1-6		
Data Space Index Support	1-6		
Data Areas	1-7		
Attached Commit Block Table	1-7		
Commit Block	1-8		
Commit Key Index	1-9		
Structure	1-9		
DATA BASE MANAGEMENT	2-1		
Introduction	2-1		
Data Sharing	2-2		
Load/Dump and Suspend	2-4		
Data Base Management Recovery and IPL	2-6		
Data Areas	2-7		
Data Space	2-8		
Data Space Index	2-12		
Key Specification Area	2-15		
User Exit Selection	2-22		
Creating a DS Index from an Existing DS Index	2-22		
Cursor	2-23		
In-Use Table	2-27		
Structure	2-28		
INDEPENDENT INDEX MANAGEMENT	3-1		
Introduction	3-1		
Create Independent Index	3-2		
Destroy Independent Index	3-2		
Find/Remove Independent Index Entry	3-3		
Insert Independent Index Entry	3-4		
Materialize Independent Index Attributes	3-4		
Modify Independent Index	3-4		
Data Areas	3-4		
Independent Index	3-4		
Index Description Template	3-5		
Structure	3-5		
JOURNAL MANAGEMENT	4-1		
Introduction	4-1		
Apply Journalled Changes	4-5		
Create Journal Port	4-5		
Create Journal Space	4-5		
Destroy Journal Port	4-6		
Destroy Journal Space	4-6		
Journal Data	4-6		
Journal Object	4-6		
Materialize Journal Port Attributes	4-6		
Materialize Journal Space Attributes	4-6		
Materialize Journalled Object Attributes	4-7		
Materialize Journalled Objects	4-7		
Modify Journal Port	4-7		
Retrieve Journal Entries	4-7		
Load/Dump and Suspend	4-7		
IPL Recovery	4-7		
IPL Synchronization	4-8		
Data Areas	4-8		
System-Wide Journal List	4-8		
Object Recovery List	4-9		
Structure	4-9		
QUEUE MANAGEMENT	5-1		
Introduction	5-1		
Recovery	5-2		
Data Areas	5-2		

Queues	5-2	Data Areas	9-6
Message Elements	5-3	Index Control Block	9-6
Structure	5-3	Structure	9-7
SPACE OBJECT MANAGEMENT	6-1	MACHINE SUPPORT FUNCTION	10-1
Introduction	6-1	INITIALIZATION/TERMINATION MANAGEMENT	10-1
Create Space	6-1	Introduction	10-1
Materialize Space Attributes	6-1	Initial Microprogram Load	10-1
Modify Space Attributes	6-1	Initial Program Load	10-3
Destroy Space	6-2	Terminate Machine Processing	10-6
Dump Space Management	6-2	Data Areas	10-6
Data Areas	6-3	VMC Communications Area (YYVCA)	10-6
Space Object	6-3	Machine Initialization Status Record (YYMISR)	10-6
Structure	6-4	Object Recovery List	10-7
INTERNAL MACHINE FUNCTION	7-1	Structure	10-7
AUXILIARY STORAGE MANAGEMENT	7-1	MACHINE CHECK MANAGEMENT	11-1
Introduction	7-1	Introduction	11-1
Invoking ASM Functions	7-2	Data Areas	11-3
Space Accounting	7-3	Machine Check Logout Buffer (RTMCLB1)	11-3
Access Group Processing	7-3	Machine Check Queue	11-3
Non-Access Group Processing	7-4	Machine Check Queue Element (RTMCQE1)	11-3
ASM Locks	7-6	Structure	11-3
Auxiliary Storage Initialization	7-7	MACHINE OBSERVATION MANAGEMENT	12-1
Storage Management Shutdown	7-7	Introduction	12-1
Directory Recovery	7-7	Materialize System Object	12-1
Data Areas	7-9	Materialize Pointer	12-1
Access Group	7-9	Materialize Pointer Locations	12-1
Free Space Directory	7-9	Trace Instructions	12-1
Permanent Directory	7-10	Cancel Trace	12-2
Temporary Directory	7-10	Trace and Cancel Trace Invocations	12-2
Access Group Member Directory	7-11	Materialize Invocation	12-2
Access Group Table of Contents	7-12	Materialize Instruction Attributes	12-2
Storage Management Vector Table	7-13	Data Areas	12-2
Sector Headers	7-13	Trace Table	12-2
Structure	7-13	Structure	12-2
MAIN STORAGE MANAGEMENT	8-1	SERVICE AND INSTALLATION MANAGEMENT	13-1
Introduction	8-1	Introduction	13-1
MSM Paging Function	8-1	Structure	13-1
Paging Function Tasks	8-2	OBJECT FUNCTION	14-1
Bring/Purge Access Group	8-11	AUTHORIZATION MANAGEMENT	14-1
Exchange Bring/Clear	8-11	Introduction	14-1
MSM Locks	8-12	Authorization Enforcement	14-2
Main Storage Initialization	8-13	Recovery	14-2
Data Areas	8-14	Data Areas	14-3
Access Group	8-14	User Profile	14-3
Permanent Directory	8-14	System Pointer	14-4
Temporary Directory	8-14	Process Control Block	14-4
Access Group Member Directory	8-14	Invocation Control Block	14-4
Access Group Table of Contents	8-14	User Profile Recovery	14-4
Lookaside Directory	8-14	Structure	14-5
Static Directory	8-14	CONTEXT MANAGEMENT	15-1
Primary Directory	8-15	Introduction	15-1
Storage Management Vector Table	8-15	Data Pointer Resolution	15-1
Sector Headers	8-15	Recovery	15-1
Paging Request Element	8-15	Data Areas	15-2
Storage Pools	8-16	Contexts	15-2
Storage Queues (Search and Change)	8-16	Name Resolution List	15-6
Structure	8-17	Encapsulated Program Architecture Header	15-6
MACHINE INDEX MANAGEMENT	9-1	Machine Communication Area	15-6
Introduction	9-1	Process Control Block	15-6
Index Structure	9-2		
Operations on Machine Indexes	9-4		

Task Dispatching Element	15-6	ERROR LOG	22-1
Structure	15-6	Introduction	22-1
RECOVERY INITIALIZATION	16-1	Data Areas	22-1
Introduction	16-1	Error Log Request	22-1
Structure	16-2	Error Log	22-2
PROGRAM CONTROL FUNCTION	17-1	Structure	22-2
PROGRAM EXECUTION MANAGEMENT	17-1	INSTRUCTION PROCESSORS	23-1
Introduction	17-1	Introduction	23-1
Program Activation	17-1	Create Instruction Processors	23-1
Program Invocation	17-2	Materialize Instruction Processors	23-1
Program De-activation	17-3	Modify Instruction Processors	23-1
Invocation Destruction	17-3	Destroy Instruction Processors	23-1
Data Areas	17-3	Request I/O Instruction Processor	23-1
Process Automatic Storage Area	17-3	Data Areas	23-3
Process Static Storage Area	17-5	Logical Unit Description (ZZSILUOB)	23-3
Structure	17-7	Controller Description (ZZSICDOB)	23-3
PROGRAM MANAGEMENT	18-1	Network Description (ZZSINDOB)	23-4
Introduction	18-1	OU/ND Table (ZZSSOUND)	23-4
Program Creation	18-1	Structure	23-5
Program Materialization	18-4	SYNCHRONOUS DATA LINK CONTROL PRIMARY	
Program Destruction	18-4	AND SECONDARY I/O MANAGERS	24-1
Program Observability	18-4	Introduction	24-1
Data Areas	18-5	System/38 Instruction Support	24-2
Program Template	18-5	Connect	24-6
Encapsulated Program	18-5	Contact for SDLC Primary	24-9
Structure	18-7	Contact for SDLC Secondary	24-9
SOURCE/SINK FUNCTION	19-1	Disconnect	24-9
ADVANCED PROGRAM-TO-PROGRAM		Test	24-10
COMMUNICATIONS STATION I/O MANAGER	19-1	Normal Flow	24-10
Introduction	19-1	SDLC Autopoll Flow	24-12
System/38 Instruction Support	19-4	Vary Off	24-13
SNA Support	19-6	Error Flow	24-13
I/O Support	19-8	Read Data Store	24-14
Error Logging	19-9	Internal Trap	24-14
Data Areas	19-9	Data Areas	24-15
Network Architecture Control Block	19-9	Link Control Block	24-15
Logical Unit Name Table	19-10	Machine-Wide Storage	24-15
Mode Table	19-10	SDLC (Synchronous Data Link Control)	
Half-Session Control Block	19-11	Input Areas	24-15
Conversation Control Block	19-11	Service Order Table	24-16
Conversation Identifier	19-11	Structure	24-16
Structure	19-11	LOCAL I/O MANAGER	25-1
BINARY SYNCHRONOUS COMMUNICATIONS		Introduction	25-1
I/O MANAGER	20-1	Internal Cleanup Routine	25-5
Introduction	20-1	Data Areas	25-6
Data Areas	20-6	Structure	25-6
Link Control Block	20-6	Console Local IOM	25-6
Service Order Table	20-6	Diskette Magazine Drive Local IOM	25-7
Controller Description Table	20-6	MFCU Local IOM	25-11
Operation Request Element	20-6	3410/3411 Magnetic Tape Local IOM	25-12
Structure	20-7	3430 Local IOM	25-14
CHANNEL I/O MANAGER	21-1	3262/5211 Printer Local IOM	25-18
Introduction	21-1	3203 Printer Local IOM	25-20
Start/Halt Device Function	21-1	LOAD/DUMP MANAGEMENT	26-1
Channel Event Processing Function	21-3	Introduction	26-1
Data Areas	21-7	Request I/O Instruction Processing	26-4
Channel Error Message	21-7	Loading Objects From a Dump Space	26-6
Channel Vary On/Off Message	21-7	Modify Logical Unit Description Processing	26-6
Structure	21-7	Error Handling	26-9
		Storage Management for Load/Dump	26-9
		Device IOM	26-9

Data Areas	26-10	SNA Support	31-4
Session Control Block	26-10	I/O Support	31-5
Dump Network Message	26-11	Error Logging	31-6
Dump Object Message	26-11	Data Areas	31-6
Load Network Messages	26-11	Station Control Block	31-6
Load Object Message	26-12	Routing Table	31-7
Recoverable Error Processing	26-12	3270 Host Field Format Table	31-8
Structure	26-13	Frame Slot	31-8
		Buffer Control List	31-8
MACHINE SERVICES CONTROL POINT	27-1	Output Request Message	31-8
Introduction	27-1	Structure	31-8
Modify Controller Description (Synchronous)	27-2		
Modify Logical Unit Description (Synchronous)	27-5	SYSTEM CONTROL ADAPTER I/O MANAGER	32-1
Modify Network Description (Synchronous)	27-6	Introduction	32-1
Modify Controller Description (Asynchronous)	27-7	Data Areas	32-4
Asynchronous Message Handling	27-9	User Message for SCA IOM	32-4
BSC/MTAM Automatic Recovery Task (BART)	27-25	Function Address Table	32-5
Data Areas	27-25	Structure	32-5
Source/Sink Active Device List	27-25		
BART Control Block	27-26	3270 EMULATION MANAGEMENT	33-1
Structure	27-27	Binary Synchronous Communications I/O	
		Manager for 3270 Emulation	33-1
MULTI-LEAVING TELECOMMUNICATIONS ACCESS		3270 Emulation Translation Function	33-5
METHOD I/O MANAGER	28-1	Data Areas	33-6
Introduction	28-1	Operation Request Element	33-6
Data Areas	28-7	Link Control Block	33-6
Link Control Block	28-7	Service Order Table	33-6
Service Order Table (SOT)	28-7	Service Order Table Address Table	33-6
Structure	28-8	Poll/Select List	33-7
		Session Line Buffer	33-7
NATIVE I/O MANAGER	29-1	Structure	33-7
Introduction	29-1		
System/38 Instruction Support	29-3	X.25 I/O MANAGER	34-1
SNA Support	29-4	Introduction	34-1
Session Control	29-6	Data Areas	34-3
I/O Support	29-7	Link Control Block (LCB)	34-3
Data Areas	29-9	Service Order Table (SOT)	34-4
Controller Description	29-9	SOT Address Table	34-4
Logical Unit Description	29-9	Request Table	34-4
Native Control Block	29-10	Receive Buffers	34-4
Routing Table	29-10	Transmit Operation Request Elements (ORES)	34-4
Lookaside Table	29-11	Trap Table	34-5
Operation Request Element and Program		Valid Send/Receive Messages	34-6
Operation Block	29-11	Structure	34-7
Function Operation Block	29-11		
Source/Sink Data Areas	29-11	SUPERVISOR AND CONTROL FUNCTION	35-1
Structure	29-11	EVENT MANAGEMENT	35-1
		Introduction	35-1
SECONDARY STATION I/O MANAGERS	30-1	Monitor Event	35-1
Introduction	30-1	Enable Event Monitor	35-2
System/38 Instruction Support	30-4	Disable Event Monitor	35-2
SNA Support	30-5	Test Event	35-2
I/O Support	30-8	Wait-on-Event	35-2
Error Logging	30-9	Retrieve Event Data	35-2
Data Areas	30-9	Cancel Event Monitor	35-2
Network Architecture Control Block	30-9	Signal Event	35-2
Logical Unit Name Table	30-10	Modify Process Event Mask	35-3
Mode Table	30-10	Recovery	35-4
Half-Session Control Block	30-10	Data Areas	35-4
Conversion Control Block	30-10	Monitor Event Template	35-4
Structure	30-11	Event Index	35-4
		Signal Event Messages	35-4
PRIMARY STATION I/O MANAGER	31-1	Process Control Block (Nonresident)	35-4
Introduction	31-1	Process Control Block (Resident)	35-4
System/38 Instruction Support	31-3		

Task Dispatching Element	35-4
Structure	35-5
EXCEPTION MANAGEMENT	36-1
Introduction	36-1
First-Level Exception Handler	36-4
Second-Level Exception Handler	36-10
Third-Level Exception Handler	36-14
Data Areas	36-17
CSEH Request Block	36-17
Structure	36-18
PROCESS MANAGEMENT	37-1
Introduction	37-1
Create Process Control Space	37-4
Destroy Process Control Space	37-4
Initiate Process	37-4
Materialize Process Attributes	37-5
Modify Process Attributes	37-5
Suspend Process	37-5
Terminate Instruction	37-6
Resume Process	37-6
Terminate Process	37-7
Create Task	37-9
Destroy Task	37-9
Data Areas	37-10
Process Control Space	37-10
Process Definition Template	37-10
Structure	37-11
RESOURCE MANAGEMENT	38-1
Introduction	38-1
Machine Support Functions	38-1
Object Serialization	38-3
Timer Services	38-11
Process Interruption	38-12
Multiprogramming Level Support	38-17
Resource Management Service Task	38-22
Resource Management Attribute Control	38-22
Access Group Control	38-25
Data Areas	38-28
Access Group	38-28
Clock Comparator Data Area (#RMCCDX)	38-30
Hold Hash Table	38-30
Hold Record Area	38-30
Lock/Unlock Input Area	38-30
Lock-Wait Data Area (#RMLKDX)	38-31
Seize/Release Input Area	38-31
Seize-Wait Data Area (#RMSZDX)	38-31
Structure	38-32
APPENDIX A. COMMON FUNCTION	A-1
Machine-Wide Storage	A-1
Destroy Object (#CFDESTO)	A-1
Get Space from IWA (#CFGIWA)	A-2
Free Space from IWA (#CFFIWA)	A-2
Object Checker (#CFOCHKR)	A-2
Report Object on Object Recovery List (#CFLOGRL)	A-2
GLOSSARY	B-1
INDEX	X-1



PURPOSE OF THIS MANUAL

This manual when used with the publications listed under *If You Need More Information* is designed to aid the IBM program support representative (PSR) to isolate a malfunction in the System/38 vertical microcode (VMC). It is assumed that the PSR has a thorough knowledge of the operation of VMC. The intent of this publication is to provide an overview of the components in VMC and a description of the functions within VMC components for recall and review purposes.

ORGANIZATION OF THIS MANUAL

VMC consists of a group of components that implement the System/38 instruction set and provide various controls and functions required to support system operation. This manual contains an introductory section to VMC, a section for each of the VMC components, and an appendix that describes the common functions that cannot be related to a component. The sections that describe the VMC components are grouped according to overall system function as described under *Vertical Microcode Overview*.

IF YOU NEED MORE INFORMATION

This manual should be used with the following publications:

- *IBM System/38 Functional Concepts Manual*, GA21-9330
- *IBM System/38 Functional Reference Manual*, Volumes 1 and 2, GA21-9331, GA21-9800
- *IBM System/38 Internal Microprogramming Instructions, Formats, and Functions Reference Manual*, SC21-9037
- *IBM System/38 Diagnostic Aids*, SY21-0584
- *IBM System/38 Service Guide*, SY31-0523
- *IBM System/38 System Control Adapter Theory-Maintenance*, SY31-0527



Vertical microcode (VMC) consists of a set of routines that implement the machine instruction set and provide various controls and functions required to accomplish a user or system task. The operations of some VMC functions, such as those that implement machine instructions, are visible to the user. Other VMC functions, such as those that manage the use of main and auxiliary storage, are not directly visible to the user.

The primary functions of VMC are categorized as follows:

- **Data Function:** Those routines that provide manipulation of user data.
- **Internal Machine Function:** Those routines that manage main and auxiliary storage and manipulate the internal indexes used by other VMC functions.
- **Machine Support Function:** Those routines that configure, initialize, and terminate machine processing.
- **Object Function:** Those routines that control addressability and access to objects.
- **Program Control Function:** Those routines that put a user program template into executable form and initialize the program prior to its execution.
- **Source/Sink Function:** Those routines that process operations involving input/output devices.
- **Supervisor and Control Function:** Those routines that establish a process and monitor the execution of processes.
- **Common Function:** Those routines that perform a variety of operations as required by other VMC and system functions.

The modules that perform these functions are grouped into sets that provide support in a specialized area. These sets of modules are called components. The VMC functions and their components are shown in Figure 0-1.

Function	Component
Data	Commit Management Data Base Management Independent Index Management Journal Management Queue Management Space Object Management
Internal Machine	Auxiliary Storage Management (ASM) Main Storage Management (MSM) Machine Index Management
Machine Support	Initialization/Termination Management Machine Check Management Machine Observation Management Service and Installation Management
Object	Authorization Management Context Management Recovery Initialization
Program Control	Program Execution Management Program Management
Source/Sink	Advanced Program-To-Program Communications Station I/O Manager (IOM) Binary Synchronous Communications I/O Manager (IOM) Channel I/O Management (IOM) Error Log Instruction Processors Local I/O Managers (IOMs) Load/Dump Management Machine Services Control Point (MSCP) MULTI-LEAVING Telecommunications Access Method I/O Manager (IOM) Native I/O Management (IOM) Primary Station I/O Management (IOM) Secondary Station and Synchronous Data Link Control I/O Management (IOM) Synchronous Data Link Control I/O Manager (IOM) X.25 Communications I/O Manager System Control Adapter I/O Management (SCA IOM) 3270 Emulation Management
Supervisor and Control	Event Management Exception Management Process Management Resource Management

Figure 0-1. Functions and Components in VMC

DATA FUNCTION

Commit Management

Commit management provides the capability to group changes to an object or set of objects within one process so the changes appear to be made simultaneously even if a system or process failure occurs before all changes are made.

Additional capability of commit management is to withdraw changes from an object or set of objects within one process and reposition the cursors to the last point where changes were committed.

Data Base Management

Data base management functions store, retrieve, update and delete data in the data base.

A data base is an area where online user data is stored and organized. Data base management functions manipulate user data and provide multiple views of the data.

Multiple views of the same data can be provided for different applications. These functions also provide for the integrity and security of data. Security is provided through the supported interfaces and authorization management of user access to the data. Integrity is provided through enforcing user-defined field descriptions.

Data stored in a data base is contained in system objects called data spaces. Each record within a data space is called an entry. All entries within a given data space have the same format. An entry can consist of a single field or an ordered collection of fields. Entries in a data space can be accessed in the sequence that they were added to the data space or in a sequence using keys.

A data space index is a system object used to access one or more data spaces through keys. A single index can cover multiple data spaces. The key can be either one field of an entry or multiple fields of an entry as specified by the user.

A cursor is a system object used to provide addressability into a data space. A cursor also provides mapping tables used to provide multiple views of the entries in a data space.

Independent Index Management

Independent index management uses a system object called an independent index. Independent indexes provide a means of storing and retrieving data by content and relative order. Independent index management supports the instructions used to insert, delete, and find index entries according to a variety of rules associated with the instructions.

Journal Management

Journal management is used to record changes made to an object along with descriptive information about the object. These changes may be simultaneously recorded on two journal spaces so, if one journal space is damaged, the information can be retrieved from the undamaged journal space. The journal, which consists of the journal port and the journal space(s), and the journaled object are automatically synchronized during IPL. This also synchronizes the journaled object with all other objects being journaled through the same journal port.

The user can place entries on the journal space, along with the entries for object changes, and retrieve the entries by a variety of search criterias.

Queue Management

Queue management allows concurrently executing processes to pass information among the processes.

A queue provides a common object that one or more processes can send information to and receive information from. Information sent to a queue is contained in a message. The sending operation is an enqueue operation. The operation that attempts to remove a message from a queue is a dequeue operation. Queues can accept a message that contains a key used to identify or sequence the messages.

Messages can be processed in first-in-first-out, last-in-first-out, or keyed sequence. Two basic types of dequeue operations are supported:

- Dequeue
- Dequeue or branch

Processes issuing a dequeue operation are placed in a wait state if the queue is empty or if there are no messages of the specified key on the queue. The dequeue or branch operation allows a process to continue processing at a specified point if no message is received from the queue. A process can specify a limit to the length of time it is to wait on a queue for a message. When multiple processes are waiting on a queue for a message, the message is given only to the first process that accepts it.

Space Object Management

Space object management provides the function used to create, materialize, modify, and destroy space objects. Space objects are used to contain any type of data. Certain special spaces are related to the execution of user programs. The spaces are explicitly created just as any other object, but are processed in unique ways by program execution management. Explicit creation occurs when an executing program issues a Create Space Object instruction.

A space object can be extended, truncated, copied, initialized, suspended, and destroyed through System/38 instructions. Space object attributes can be materialized and modified.

INTERNAL MACHINE FUNCTION

Storage Management (Auxiliary and Main)

Storage management enables a program to access another program, VMC data object, and other user objects as if they were residing in a single address space. The space is addressable through 6-byte virtual addresses. The programs and objects are permanently stored on auxiliary storage that consists of nonremovable disk units. These programs and objects can be executed or manipulated only when they (or a portion of the programs and objects) are in main storage. Storage management allocates and maintains data on auxiliary storage and provides copies of this data in main storage as required.

Storage management consists of two components:

- Auxiliary storage management (ASM) that allocates storage, maintains directories, and maps virtual addresses to auxiliary storage locations. ASM also manages space allocation for access groups.
- Main storage management (MSM) that transfers data (including access groups) to and from main storage and manages storage resources (pages).

Storage management services such as page transfers, access group manipulations, and partitioning of storage into storage pools are invoked by the resource management routines.

Following are the units of storage used by storage management:

Page: A 512-byte block of storage. This is the basic unit for storage management operations.

Sector: A 520-byte auxiliary storage record that contains a page and an 8-byte storage management header.

Segment: A contiguous address space that contains up to 128 pages (64 K bytes).

Segment Group: An address space that can contain up to 256 consecutive segments (16 MB).

There are two types of segment groups:

- Permanent: A segment group that exists until it is explicitly destroyed.
- Temporary: A segment group that is automatically destroyed at initial microprogram load (IMPL).

Access Group: An access group is a system object that collects temporary objects into a group that can be operated on as a single unit by storage management. An access group is created with a segment identifier and is allocated a block of contiguous space on auxiliary storage. Other system objects can be allocated within this block, each object having an identifier allowing each object to be accessed individually. However, special directory information enables storage management to transfer all objects within the access group to and from main storage as a single unit.

Virtual Storage Addressing

Several different types of addresses exist in the System/38. The machine processor supports only a 6-byte address that contains the following:

- Segment identifier (4 bytes)
 - Offset into the segment (2 bytes)
- VMC views this address as follows:
- Segment group identifier (3 bytes)
 - Offset into the segment group (3 bytes)

Because the machine processor supports only a 2-byte offset, VMC must process overflows into the segment identifier. This is done explicitly by calculating the offset or implicitly by invoking the effective address overflow exception handler.

One bit of the segment group identifier indicates if the segment group is permanent or temporary. A second bit indicates if the segment group is allocated within an access group. The remaining 22 bits are used to identify the segment group.

When a segment group is created, storage management assigns an 8-byte address to the new segment group. This address consists of the following:

- Segment group extender (2 bytes)
- Segment group identifier (3 bytes)
- Offset into the segment group (3 bytes)

The segment group extender is a 2-byte extension to the 6-byte VMC address. The extender together with the 3-byte segment group number forms a 5-byte identifier that is assigned only once during the life of the machine. The 3-byte segment group identifiers are assigned sequentially until hex 3FF000, at which time the segment group extender is incremented and the address regeneration routine must be executed. At any one time, each 3-byte segment group identifier in the system is unique, although once the corresponding segment group is destroyed, the identifier is available for reassignment.

The segment group extender is used only by VMC routines. When an object is accessed by a pointer instruction, the extender in the object header is compared with the extender in the 16-byte system pointer. If they do not match, the pointer is invalid (the pointer in this case is an old pointer to a destroyed segment with a reassigned segment identifier). The extender is not used for internal VMC addresses because these addresses are checked for validity and written with a special identifier if they are no longer in use.

Each segment group in main storage begins with a 32-byte header. The contents of this header are as follows:

- Segment group type (1 byte)
- Flag byte (1 byte)
- Size (2 bytes)
- Extender (2 bytes)
- Object base segment address (6 bytes)
- Reserved (14 bytes)
- Space locator (6 bytes)

The flag contains the object existence bit. This bit is used to reclaim unused storage if a system failure occurs when an object is being created. The existence bit is set to 0 by a create segment operation. Any module that invokes the create segment group function to create a permanent object sets this bit to 1 and writes the first page of the object to auxiliary storage after the object is created. Any module that creates a temporary object also sets this bit to 1 but does not write the object to auxiliary storage because temporary objects are not recovered after a system failure. As part of recovery after a system failure, any segment group that does not have the existence bit set to 1 is destroyed.

As part of object creation, the creating module also stores the address of the primary segment group of an object in the header of each segment group of the object. An object that consists of a single segment group points to itself. As a part of the reclaim function, any segment group that points to a nonexistent primary object is also destroyed. This ensures that all segments of a multisegment object are destroyed in the event of a system failure that occurred during object creation.

Machine Index Management

Machine index management provides an indexing function that is used by other VMC components. Machine indexes consist of binary radix trees that are used to store and retrieve data. Machine index management provides a variety of functions that allow other VMC components to insert, retrieve, and remove entries from a machine index.

MACHINE SUPPORT FUNCTION

Initialization/Termination Management

The initialization function of VMC is used to put the system into a state in which instructions can be executed. The IMPL function is activated using the machine power-on sequence or the system console. Once the machine has been initialized, the machine-to-programming transition function provides the user of the machine interface with the capability of initiating a user process. User processes can be initiated from the data that exists either on the primary load/dump device or within the machine on auxiliary storage.

The terminate machine processing function provides the user of the machine interface with the capability of destroying all processes in the machine and of either turning off the machine power or putting the machine into a checked-stop state.

Machine Check Management

Machine check management provides the functions that report machine malfunctions to the machine interface through function check exception, machine check exceptions, and machine check events. The functions performed in machine check management are as follows:

- Initiate recovery where possible
- Record machine malfunctions
- Signal malfunctions to the machine interface
- Signal that machine execution is terminated

When a malfunction occurs, the machine saves related data, reactivates the user processes that were active at the time of the failure, and signals the malfunction to the machine interface as an exception or an event. Execution of the instruction in progress at the time of the machine check does not complete and the user handles the exception in the same manner as any other exception.

If the malfunction is unrecoverable, machine execution terminates and indicators on the operator/service panel come on to indicate the condition.

A process must monitor the function or machine check exception if the process is to be notified of either a function or a machine check that occurred in the execution of a System/38 instruction.

Machine Observation Management

Machine observation management provides the functions to observe the activity of the machine. Observation is provided at the machine interface through the use of trace and materialize instructions. These instructions can monitor the execution of programs and observe the execution of System/38 instructions.

The functions provided by machine observation management are as follows:

- Trace the execution sequence of a user process by monitoring calls and returns.
- Trace the execution sequence of a user program by monitoring instruction execution.
- Materialize the addressability of pointers.
- Materialize information about system pointers.
- Materialize the current assignment of program objects.
- Materialize the location of pointers in a space.

Service and Installation Management

Service and installation management is used in problem determination and to modify or configure the machine attributes. The functions provided are as follows:

- Display/alter/dump
- Virtual storage stand-alone dump
- VMC trace
- Print stand-alone dump
- Address stop/instruction step
- Machine configuration update facility
- Auxiliary storage initialization
- Link/loader
- VMC error log facilities

Refer to the *System/38 Diagnostic Aids* manual for information concerning the use of these functions.

OBJECT FUNCTION

Authorization Management

Authorization management provides the functions defined through the machine interface to control the use of user objects, system resources, and privileged System/38 instructions.

A user profile is the collection point for authorization related information that is defined at the machine interface and monitored by VMC. Security functions provided above the machine interface are monitored by VMC.

Every process in the system executes under control of a user profile. This allows both VMC and Control Program Facility (CPF) to monitor the activities of each executing process. The information in the user profile sets limits as to what the executing process can and cannot do. A given user profile can be unique to one user or can be shared among several users.

Context Management

Context management is used to store addressability to objects in the system and to transform a reference by a symbolic name into an address. Addressability to a system object can be placed in a context such that the context can be used to locate the system object using a symbolic address. The symbolic address of an object used for context addressing consists of an object type, subtype, and name. The symbolic address of each object addressed by a single context must be a unique address within that context.

Recovery Initialization

Recovery initialization is a recovery common function used by VMC components to recover objects at IMPL time. Recovery initialization builds a machine index with an entry for every base segment of an object and every secondary segment for multiple segment objects. Recovery initialization also builds an index containing every user profile. This index is used by authority initialization. Recovery object read does selective finds on the base segment index allowing base segments and/or secondary segments to be retrieved by object type or secondary segment type.

PROGRAM CONTROL FUNCTION

Program Execution Management

Program execution management controls the synchronous execution of instructions within translated programs. This is accomplished through a call/return instruction set that standardizes interprogram linkages and allows applications to be built from a combination of high-level language programs, compiled control language programs, and programs coded using the System/38 instruction set (program templates).

The Call and Return instructions provide a common linkage convention (saving and restoring of registers, and so on) between any compiled high-level language program, compiled control language programs, and program templates. Because high-level languages and control language programs are compiled into a program template, their calls and returns are compiled into a common call/return interface. Call and Return instructions are the only means of transferring control between separate (external) programs within a process; control can be transferred between inline (internal) programs using other instructions (Branch, Call Internal).

Program activation consists of putting the program into an executable state within a process. The results of program activation are the allocation and initialization of static storage areas in the program. Program invocation causes the flow of control within the process to be passed to the entry point of the instruction stream where execution is to begin. The results of program invocation are the suspension of the execution of the invoking program and the allocation and initialization of the automatic storage defined in the invoked program.

Program execution management also provides a program activation function. An activation is a logical copy of the static objects of a program. An activation is established explicitly by an Activate Program instruction and implicitly during the invocation of a program that has not been activated.

The logical objects in an invocation include activation addressability, automatic object (program objects allocated at invocation and deallocated at return), exception handling specifications, and the call/return relationship specifications. Invocations are created through a call, transfer control, event detection, exception detection, initial program in a process phase, or a data base user exit program.

Program Management

Program management provides the functions that are used to create executable programs, materialize the attributes of a created program, and destroy created programs. The create function consists of converting the program template that is received through the machine interface into a form that is executable in the system. This function, called translation or encapsulation, syntax checks the program template, generates a system object called a program, returns diagnostic information as required, and returns a system pointer to the generated program system object.

The materialize function returns a copy of the input program template. The destroy function removes addressability to the created program and frees the resources associated with the program to be destroyed.

SOURCE/SINK FUNCTION

Instruction Processors

Instruction processors provide support for the System/38 instructions that operate on the following objects:

- Logical unit descriptions (LUDs)
- Controller descriptions (CDs)
- Network descriptions (NDs)
- Request I/O response queue

These processors support instructions that create, materialize, modify, and destroy the preceding objects and the Request I/O instruction.

Machine Services Control Point

Machine services control point (MSCP) supports other source/sink functions by allocating and controlling all source/sink resources. Services include establishing physical and logical paths over which a user can communicate with an input/output device, allocating queues, and creating the tasks necessary to establish a session with a device.

I/O Managers

I/O managers (IOMs) interface with the horizontal microcode (HMC) tasks associated with the physically attached I/O devices. IOMs are either device or function-dependent and execute as VMC tasks. Both the routines and the tasks under which they execute are referred to as IOMs. There is an IOM for each I/O device or I/O hardware capability as follows:

- Binary synchronous communications IOM (for binary synchronous communications lines)
- Channel IOM (internal)
- Error log (internal)
- Synchronous data link control primary line IOM (for synchronous data link control communications lines)
- Synchronous data link control secondary line IOM (for synchronous data link control communications lines)
- Local IOM (for console, printer, card, diskette, and tape devices)
- Load/dump (for load/dump operations)
- MULTI-LEAVING telecommunications access method IOM
- Native IOM (for locally attached work stations)
- Synchronous data link control secondary or APPC station IOM (for systems network architecture [SNA])
- Synchronous data link control primary station IOM (for remotely attached work stations)
- System control adapter IOM (internal)

Source/Sink Data Areas

Source/sink components use several common data areas for communications with other components. A description of these areas is provided here instead of with the individual source/sink components to eliminate duplication.

Machine Configuration Record (MCR): The MCR contains information about the physical devices that are locally attached on a given machine. The MCR is built during machine manufacture and is updated whenever devices are installed or removed from the system. Source/sink components use the MCR to determine how such fixed resources as I/O registers and channel priority are allocated to the devices. The MCR resides in a preallocated segment built at installation time.

Source/Sink Active Device List: The source/sink active device list is a dynamically built control block used and controlled by the MSCP. It contains information about the current status of all active devices. It is also used to record the allocation of system resources such as tasks, queues, and switched network lines. The source/sink active device list has a base that resides in a preallocated segment built at installation time.

Source/Sink OU/ND Table: The source/sink OU/ND table is a control block used by the source/sink instruction processors to keep track of how many NDs exist for each OU and which ND(s) are varied on or in diagnostic mode. The Create ND instruction uses the table to determine if another ND can be created. The Modify ND instruction uses the table to check if an ND can be varied on or set in diagnostic mode. During IMPL the table is built in a temporary segment, and its address is placed in the source/sink active device list.

Feedback Record: The feedback record is the message that is enqueued to the appropriate user response queue to signal the completion of a function that had been initiated by a Request I/O instruction. It contains a return code or status indicators as well as device-dependent data generated in response to the associated request.

IOM Queue: Each IOM has a single input queue on which it receives requests to do work by Request I/O instructions, responses from the operational unit (OU) task, and control messages from other source components. An IOM queue is allocated when the corresponding IOM task is created and is always used exclusively by that IOM task.

Operational Unit (OU) Queue: An OU queue is the input queue on which an operational unit microtask receives operation request elements (OREs) from the IOM. Each OU queue serves a specific OU task.

Queue Control Table (QCT): The QCT is a data structure in the machine nucleus that provides information to the table-driven OU task. A QCT exists for each OU in the system. The QCT contains the I/O register assignments for the device and dynamic information to control the OU task (for example, wait for the device to complete, get the next command, and terminate the current request). QCTs for all devices are allocated and built from information in the MCR at link/load time.

Operation Request Element (ORE): An ORE is a send/receive message used to request services from an OU task. An ORE is built by an IOM and sent to an OU queue. Functions requested include channel operations, requests to be sent to the device adapter, and control information.

Send/Receive Messages (SRM): An SRM is a general term for all the messages sent between the components that support the source/sink function. An ORE is an example of one of these message types.

SUPERVISOR AND CONTROL FUNCTION

Event Management

Event management provides the functions that enable a user to monitor for the occurrence of a set of events and to take action based on the event or events that occurred.

Events relate to and define occurrences that happen within the machine. These events can be of importance to a particular user application. Event management allows a process to monitor for an event. The activity being monitored can represent conditions both internal and external to the monitoring process. That is, one process can be monitoring conditions caused by the same process or by other processes that can be concurrently in the system.

Events are classed as machine-wide or process-only. When a machine-wide event is signaled, an event monitor index is searched to locate the processes that are monitoring for the event. The process establishes a monitor for an event before the event occurs when a process-only event is signaled.

Events can be signaled either from a user process or from the machine. User signaled events are generated as a result of the System/38 Signal Event instruction. These events are defined outside the scope of the hardware and have no meaning to the hardware. Monitoring of these events is based on user protocol.

Machine events are defined as part of the machine and are detected and signaled by the machine. Machine events are classed as either object-related or machine-related. Object-related events are signaled when an object has been created or destroyed, or when the attributes of an object are modified. Machine-related events are signaled when conditions occur in the machine that are not directly related to a system object (for example, machine resource limit exceeded).

The intention to monitor for the occurrence of an event is specified by establishing an event monitor within a process. Event monitors describe to the machine what event or condition is to be monitored and the action that is to be performed when the event occurs.

Exception Management

Exception management provides the functions that enable a user to monitor for the occurrence of an exception and to process an exception when one occurs.

Exceptions are either errors that are detected by the machine or conditions that are detected by user programs. Exceptions detected by the machine occur as a result of the execution of a System/38 instruction and are signaled implicitly. Exceptions detected by a user program occur as a result of some condition detected within the program and are signaled explicitly by using the Signal Exception instruction.

Exceptions are monitored through an exception description. An exception description is a program object that defines the exceptions that are to be monitored and the action that is to be performed when the exception occurs.

Process Management

Process management supports the structure that enables the concurrent processing of work in the System/38. This structure, called a process, enables programs to be executed.

Process management initiates and terminates processes, displays and modifies the attributes of processes, and suspends and resumes the execution of processes. A process can be independent or a subprocess (dependent).

A user process is built as a result of an Initiate Process instruction. This process is assembled from information contained in the process definition template (PDT). The main part of a process is the process control space (PCS) that is created as a result of a Create Process Control Space instruction. When a process is initiated, a task dispatching element (TDE) is built on the first PCS segment.

A process control block (PCB) is then built. The PCB describes the process and is built in two parts. A resident PCB is built in the first PCS segment. The resident PCB contains fields that may have to be referenced asynchronously by other processes. The remainder of the PCB is built in the second segment.

The process invocation work area is then built in the second segment following the PCB. The process invocation work area is a block of storage, managed by VMC as a stack, that is used for automatic storage required by VMC routines and for the invocation control blocks for all current program invocations under an executing process.

A user process is the basic unit (task) of dispatchable work in the system. Two other types of tasks exist in the system:

- A microtask that executes HMC instructions. Such tasks are normally associated with I/O devices.
- A VMC task that is created to perform work asynchronous to user operations.

The microtask and VMC task are also supported by process management.

Resource Management

Resource management controls the allocation and management of the resources required during the execution of processes in the system and calculates the amounts of resources used by the processes. Resource management controls the following types of resources:

- Storage
- Processor
- System objects (such as devices and data space)

Because several processes can be executing concurrently (multiprogramming), the resources must be controlled to manage contention for the use of resources among the executing processes. Based on limits established by the system administrator for the processes in the system, resource management grants the resources requested by processes within the limits and priorities defined for a user environment.

Resource management can be viewed from several levels. To the user above the machine interface, resource management allocates objects to application programs, provides storage and processor usage as needed, and ensures the integrity of data and the execution environment. To the CPF or the user control program, resource management supports scheduling, allocations, and the execution of processes. Within VMC, resource management performs the immediate distribution of resources to efficiently satisfy processing demands.

Resource management is accomplished through control and monitoring functions. Control functions are provided through:

- System/38 instructions that:
 - Specifically request the allocation of resources (locking instructions)
 - Control the overall level of work in the system and distribute the resources (modify instruction)
 - Provide better use of storage resources (access group and access state instruction)
- Process attributes that:
 - Affect the distribution of resources
 - Limit the use of resources
- Object attributes that affect the performance of the users of the object

Monitoring functions are provided through:

- System/38 instructions that provide information on the use of and contention for resources (materialization instructions)
- Process attributes that provide information relative to the use and allocation of resources for a process
- Machine events that indicate that a user has exceeded the specified resource limit
- Exceptions that indicate that a resource is not available or that the use of a resource has exceeded a specified limit

COMMON FUNCTION

Common function modules provide a variety of functions for the VMC components. Most common function modules that perform functions that are directly related to an existing VMC component are described within that component. Common function modules that cannot be directly related to a VMC component are described in *Appendix A* of this manual. Refer to the *Contents* for a list of the functions described in the appendix.

Relationship of Components

VMC components are made operational as a result of an alternate IMPL or an IMPL operation. The alternate IMPL operation is a stand-alone operation that is used to load the IMPL functions into the system from an external device. Once the alternate IMPL operation is complete, the IMPL functions can be initiated to make the system operational.

Note: For a detailed description of the alternate IMPL and IMPL operations, refer to the *System/38 System Control Adapter Theory-Maintenance* manual.

Alternate IMPL

The purpose of an alternate IMPL operation is to read the contents of a diskette magazine, execute the test and load operations contained on the diskette magazine, initialize certain areas of main and auxiliary storage, and establish a system capable of executing some functions. This is accomplished by selecting alternate IMPL on the rotary switches on the service panel and starting a load operation. This starts the execution of some microcode instructions that are stored in read-only storage of the system control adapter (SCA). These microcode instructions cause the directory information, test procedure, load procedures, and other microcode procedures to be stored or executed in the system as appropriate.

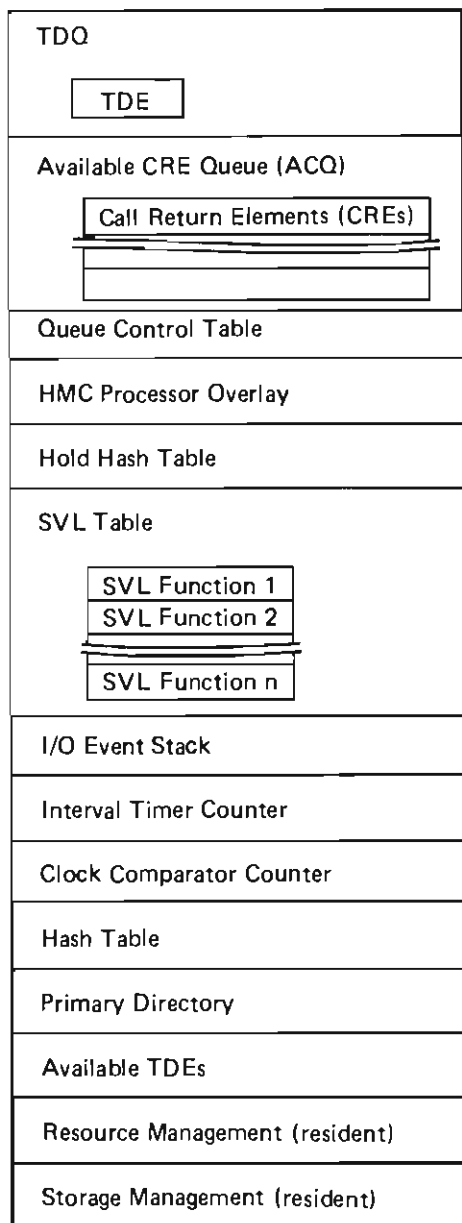
When the SCA microcode has completed, the service monitor and the resident portion of the HMC are loaded in the system. At this point, the system is capable of executing some diagnostic functions and a set of VMC functions. The VMC functions that can be executed consist mainly of the following:

- **MCR update:** This function provides the capability of modifying the MCR. The MCR defines the hardware configuration of a system.
- **Auxiliary storage initialization:** This function initializes auxiliary storage and builds the storage management directory used to map auxiliary storage space.
- **Link/loader:** This function copies the microcode from the diskette magazine to the area reserved for this code during auxiliary storage initialization.

Internal Microprogram Load

The operation of IMPL is similar to alternate IMPL except that the VMC nucleus is loaded instead of the service monitor, and that the system is capable of executing the System/38 instruction set at the end of initialization.

Figure 0-2 shows a logical view of the main areas in the VMC nucleus. The figure also shows that the task dispatching queue (TDQ) contains a task dispatching element (TDE) for storage management initialization. At the end of the IMPL sequence, HMC dispatches this queue and control is passed to the storage management initialization routines. When these routines have completed execution, VMC is fully functional and capable of supporting the System/38 instruction set.



These areas are required for HMC instruction execution.

Vertical Microcode and the System/38 Instruction Set

Vertical microcode consists of the following:

- A component that converts the System/38 instructions in a user program into executable form
- A group of components that interpretively execute portions of a user program
- A group of components that provide supervisory and support functions

The components of VMC implement the System/38 instruction set in a manner similar to that of the microcode in conventional systems. In the System/38, however, VMC must first convert the System/38 instructions into internal microprogram instructions. The internal microprogram instructions, in turn, cause the HMC and the hardware to execute the functions that were requested in a user program.

Figure 0-3 shows an example of what could take place when a user program is executed on the system. Notice that the VMC components are between the machine interface and the HMC and the hardware. Also note that the functions below the machine interface are transparent to the user programmer.

First, the user program must be converted from System/38 instructions into internal microprogram instructions. This is accomplished by the translator, a function of program management. This conversion is also called encapsulation. Next, the program must be made ready for execution. This is accomplished by activating, then invoking, the program. Activation initializes the static storage areas used by the program, and is initiated either by the Activate Program instruction or implicitly by an invocation. Invocation causes the automatic storage areas used by the program to be initialized and the program to be executed. Activation and invocation are performed by program execution management.

Figure 0-2. VMC Nucleus

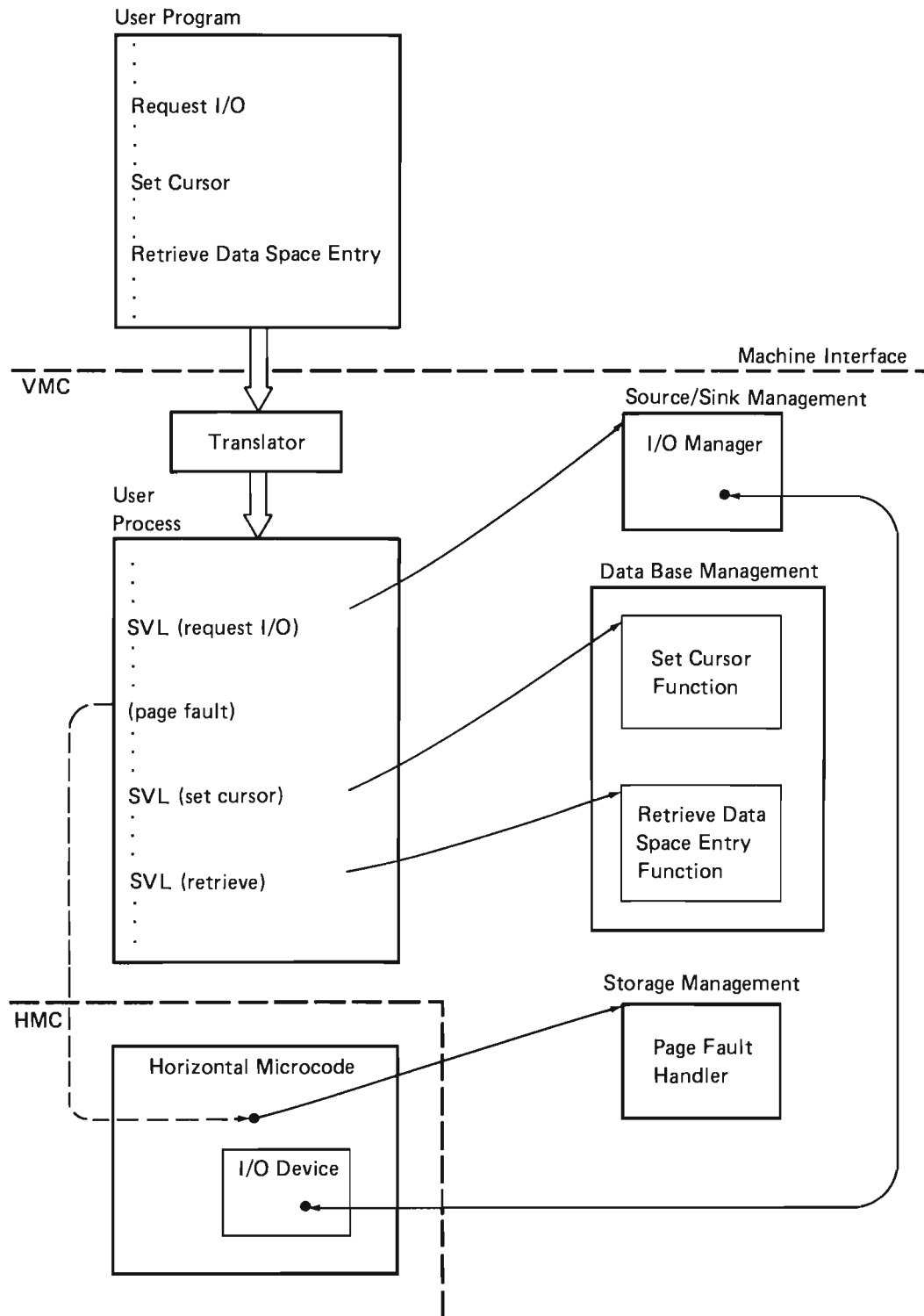


Figure 0-3. Program Execution

Program execution occurs in an environment called a process. Processes are similar to tasks on other systems. Several processes can be executing concurrently in the system; the execution of these processes is controlled by process and resource management. The processes can also be competing with each other for system resources (for example, storage or processor time); these resources are allocated and controlled by resource management.

A program, during its execution, may require the use of one of the interpretive VMC functions. For example, a program may need data from an I/O device. When this program was encapsulated, internal microprogram instructions were inserted into the instruction stream. These internal microprogram instructions establish the supervisor linkage (SVL) to the VMC code that processes the Request I/O instruction.

Figure 0-4 shows an example of a Request I/O operation. The Request I/O instruction processor analyzes the request, and builds and sends a message to the appropriate IOM. The IOM is a VMC task executing device-dependent code to service a specific I/O device. When the IOM is ready to accept more work, the request I/O message is dequeued from the IOM queue and processed. The IOM generates the specific device commands necessary to perform the requested functions and sends an ORE to the OU task. The OU task consists of the HMC functions required to interface to the channel hardware to start an I/O operation to the device. When the device completes the operation, the OU task puts the completion status in the ORE and sends it back to the IOM queue. The IOM in turn sends a feedback record to the response queue indicating that asynchronous processing of the I/O request is completed.

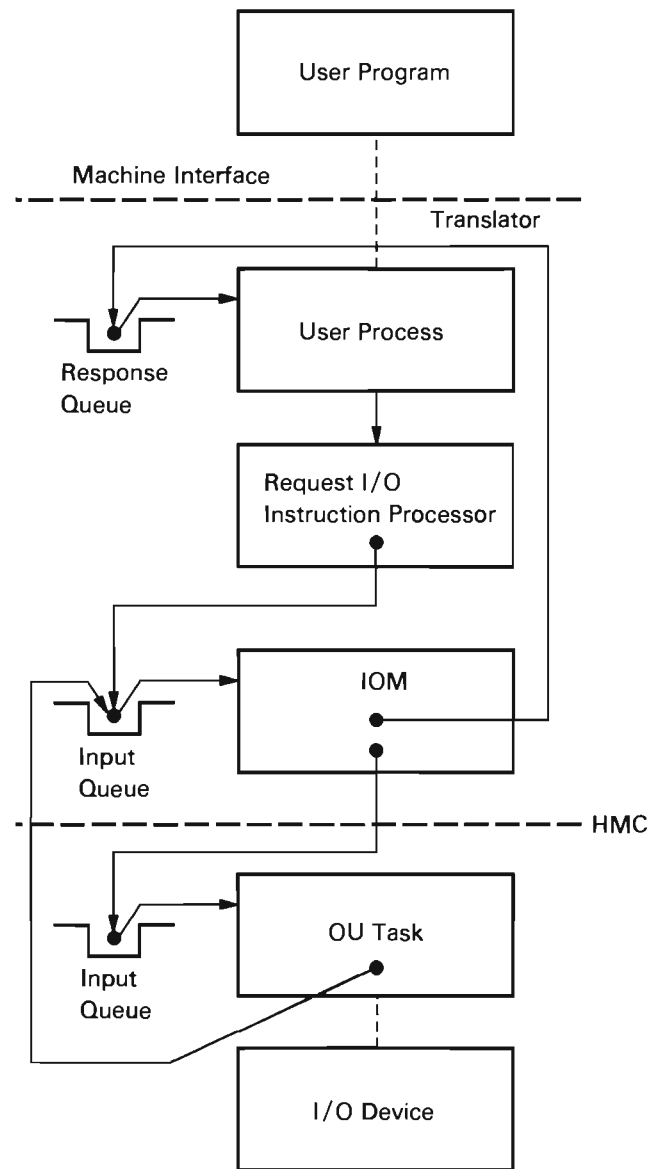


Figure 0-4. I/O Operation

Because the I/O functions operate asynchronously, the user program can continue executing after it issues the Request I/O instruction. Before using the data areas associated with the I/O request, it is necessary to dequeue the feedback record to assure the asynchronous I/O processing is complete. At any time during execution of a program, reference can be made to either executable code or data at a virtual address that is not in main storage. In this case, storage management receives control through an internal microprogram exception that was generated as a result of the page fault. Storage management executes the necessary functions to transfer the data from auxiliary storage. I/O operations to auxiliary storage files are controlled entirely by storage management. They do not use the source/sink or IOM support. When storage management has completed the operation of bringing the referenced virtual storage into main storage, the user program continues executing. The user is not aware of the paging operation and can assume all allocated storage is directly accessible to the program.

Commit Management

INTRODUCTION

Commit management provides the capability to group changes (commit) to an object or set of objects within one process so the changes appear to be made simultaneously. This is guaranteed even if a system or process failure occurs before all changes are made. If a system or process fails, any group of incomplete (uncommitted) changes are automatically withdrawn from the object(s).

Commit management also provides the additional capability to withdraw (decommit) changes made to an object or set of objects within one process and reposition the cursors to the positions prior to where the group of changes were withdrawn.

Commit management supports the following System/38 instructions:

- Commit
- Create Commit Block
- Decommit
- Destroy Commit Block
- Materialize Commit Block Attributes
- Modify Commit Block

Transactions Under Commitment Control

A transaction under commitment control starts with the access of the data base by a cursor and ends with the execution of either a Commit or a Decommit instruction. If neither of these instructions are executed because of a program or system failure, decommit is assumed and any changes to the data base made by the transaction are withdrawn.

When a transaction ends with the execution of a Commit instruction, the results are:

- All data space index keys reserved during the transaction are freed.
- All record locks held during the transaction are freed.
- All changed records are made available for further updates.

When a transaction ends with the execution of a Decommit instruction, the results are:

- All changes made by the transaction are withdrawn.
- All data space index keys reserved during the transaction are freed.
- All record locks held during the transaction are freed.
- All records are made available for updating.
- All cursors under commitment control are reset to the position they had at the start of the commit cycle.

If a program or system failure occurs, decommit is implicitly performed as part of process termination or as part of a subsequent IMPL.

Changes Under Commitment Control

Changes made under commitment control are not held until the end of the transaction but are applied immediately to the data base.

The commit block and all data spaces that are changed under commitment control must be journaled to the same journal port. The images of the records before they are changed are always journaled. If it is necessary to withdraw a transaction, the image of the records before they are changed are placed back in the data base.

Record Locking

All records that are changed or added to by a transaction are locked until the transaction completes. In addition, locks can be optionally held until the transaction is complete on all records accessed by the transaction regardless of whether the records are changed. This prevents the record from being changed by another transaction during the duration of this transaction.

Data Space Index

For unique data space indexes, any key removed by a transaction is reserved until that transaction is completed. While a key is reserved, no other transaction can add that key to the data space index. This allows all transactions to be withdrawn at any time prior to executing a Commit instruction.

IMPL Recovery

The IMPL recovery for commit is actually a cooperative effort of three VMC components, data base, journal, and commit. The sequence performed in IMPL recovery is as follows:

1. The data base component recovers data spaces and data space indexes.
2. The journal component recovers journal ports and journal spaces.
3. The journal component, module #JOISYNC, synchronizes the journaled objects with their journal ports. The synchronization is accomplished by scanning the journal spaces and, for each journal entry, calling the component that controls the journaled object to verify that the change to the object (represented by the journal entry) is present in the object.
4. The commit component, module #COINIT1, scans the attached commit block table and, for each commit block:
 - a. validates the fields in the commit block
 - b. decommits any uncommitted changes
 - c. places an entry on the object recovery list giving the status of each commit block
5. The data base component recovers the data spaces that were affected during IMPL recovery by the journal and commit components.
6. The commit component, module, #COINIT2, initializes the attached commit block table.

System/38 Instruction Support

Commit

The Commit instruction processor (**#COCOMIT**) is invoked by the supervisor link (SVL) router as a result of a Commit instruction. **#COCOMIT** places a commit entry containing a description of the commit on the journal and decreases the journal in-use counter. The commit description is also placed in the commit block. Any data space index keys that were reserved for the commit block are removed and any data space entry locks held by the commit block are released along with the data space entry locks held by all cursors under commitment control. For any data space that had its in-use count in the data base in-use table increased during this commit cycle, the in-use count is decreased. Any data space previously locked by a Commit instruction for shared update is released.

Create Commit Block

The Create Commit Block instruction processor (**#COCRCOB**) is invoked by the SVL router as a result of a Create Commit Block instruction. **#COCRCOB** creates a permanent object called a commit block. The commit block contains information concerning the changes to objects under control of commit management. Some of the data contained in the commit block can be materialized through the Materialize Commit Block Attributes instruction.

Decommit

The Decommit instruction processor (**#CODCMIT**) is invoked by the SVL router as a result of a Decommit instruction. When **#CODCMIT** is invoked, all uncommitted changes are withdrawn from the object causing all data space images to be restored to their previous state. All indexes are simultaneously maintained. If the previous state of the index cannot be achieved, the index is invalidated and the decommit cycle continues.

The changes necessary to perform the Decommit instruction are journaled with an entry subtype indicating the change is because of a Decommit instruction. A decommit entry is then placed on the journal. This entry contains the decommit status. Any data space index keys that were reserved to this commit block are removed.

The data space entry locks held by the commit block and all cursors under commitment control are released.

The position of the cursors under commitment control is reset to the position the cursors had when the commit cycle started (when the start commit entry was journaled). Cursors that were placed under commitment control after the commit cycle started are reset to the position they had when placed under commitment control. Cursors that were removed from commitment control before the decommit cycle are not repositioned.

Each data space modified by the Decommit instruction is forced to auxiliary storage. For each data space that had its in-use count increased during this commit cycle, the count is decreased. For any data space that was locked by the Commit instruction, the lock is released.

The journal in-use count is then decreased.

#CODCCF is invoked to perform the decommit process if at process termination there is a commit block attached to the terminating process that started a commit cycle (start commit) but has not completed. **#CODCCF** is also invoked if at IMPL time there is a commit block attached to any process from a prior IPL that started a commit cycle.

Destroy Commit Block

The Destroy Commit Block instruction processor (**#CODSCOB**) is invoked by the SVL router as a result of a Destroy Commit Block instruction. This module destroys any selected commit block that is not attached to a process.

Materialize Commit Block Attributes

The Materialize Commit Block Attributes instruction processor (**#COMACOB**) is invoked by the SVL router as a result of a Materialize Commit Block Attributes instruction. This module returns either the commit block creation template with the current commit block attributes, or the commit block status including the number of uncommitted changes, the number of objects under commitment control, and the commit description of the last successful commit.

Modify Commit Block

The Modify Commit Block instruction processor (#COMOCOB) is invoked by the SVL router as a result of a Modify Commit Block instruction. #COMOCOB performs the following:

- Attaches a commit block to a process.
- Detaches a commit block from a process.
- Places objects (cursors) under commitment control.
- Removes specific objects (cursors) from commitment control.
- Removes all objects (cursors) from commitment control.

A commit block can be attached to a process as long as it is not already attached to that process or to any other process. Once a commit block is attached to a process, the commit block can only be modified by that process. An entry is placed on the journal indicating that the commit block is attached.

A commit block cannot be detached from a process if a start commit was journaled and no ensuing Commit or Decommit instruction was executed, or if there are any objects still under commitment control (in the commit object list). An entry is placed on the journal indicating that the commit block is detached.

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 are not under commitment control because there may be another cursor, not under commitment control, over the same data spaces and changes made by this cursor would not be under commitment control.

All objects changed under the control of a given commit block must have their changes journaled to a single journal port and that journal port must be the same journal port to which the commit block is journaled.

When the cursor is placed under commitment control it must be eligible for commitment control.

Any cursor that is to be removed from commitment control must be eligible for removal from commitment control. No journaling is performed when removing a cursor from commitment control.

Journal Support

Journaling of the Commit Block

The commit block must be a journaled object when it is attached to a process. The following commit block activity is journaled:

- **Modify Commit Block:** Modify Commit Block (attach) instruction and Modify Commit Block (detach) instruction are journaled. The journaled entry contains no data.
- **Start Commit:** This entry is placed on the journal implicitly. The Commit, Set Cursor, and Insert Data Space Entry instructions cause a start commit entry to be journaled. After the start commit entry is journaled, the journal sequence number of the entry is placed in the commit block. This sequence number is used to limit the search of the journal when a Decommit instruction is executed. The journaled entry contains no data.
- **Commit:** When a Commit instruction is executed, the commit block is journaled. The journaled entry contains the commit description entry.
- **Save Cursor Position:** The first time within a commit cycle that a Set Cursor instruction is executed for a given cursor, the data necessary to restore that cursor's position is journaled. If a Decommit instruction is executed, the data is used to reposition the cursor.
- **Decommit:** When a Decommit instruction is executed, the commit block is journaled. The journaled entry contains the decommit status of the commit block.
- **Destroy Commit Block:** When a Destroy Commit Block instruction is executed, the commit block is journaled.

Journaling of Changes to Data Spaces

Because the journal is used to withdraw the changes in case of a process/system failure or if a Decommit instruction is executed, the before image of each change made under commitment control is journaled. Journal entries for data space changes made under commitment control contain an additional journal entry prefix field called commit ID. Commit ID is the sequence number of the start commit journal entry.

Journal Space Use-count

Because the journal is used in the process of decommitting, journal receivers containing uncommitted changes cannot be suspended, destroyed, or restored. This is controlled by maintaining an in-use count with the journal receiver. The in-use count is increased when a start commit entry is journaled and is decreased when a Commit or Decommit instruction is executed. An exception is signaled if an attempt is made to suspend, destroy, or restore a journal receiver with a nonzero in-use count.

IMPL Synchronization of the Commit Block with the Journal

Before the commit component IMPL recovery (#COINIT1) runs, the journal component IMPL synchronization phase synchronizes all commit blocks with the journal. When the journal component finds a commit block, #COJOSYN is invoked to perform synchronization.

Data spaces containing uncommitted changes are also synchronized with the journal. The journal component passes these journal entries to the data base component. There are no special considerations made for commit during this phase.

Cursor Support

Activate Cursor

The EPAHCOMT flag in the cursor header must be off before the cursor is activated. The flag has no meaning if the cursor is not activated.

Placing a Cursor Under Commitment Control

The cursor is placed under commitment control by the Modify Commit Block instruction. The cursor must be active within the process and can not hold any data space entry locks. All of the data spaces under the cursor must be journaled to the same port as the commit block.

Removing a Cursor Under Commitment Control

The cursor is removed from commitment control by the Modify Commit Block instruction. The cursor must not hold any data space entry locks. If there are any uncommitted changes made by the cursor, the data spaces containing the changes are placed in in-use mode (data base in-use count is increased) by commit. The in-use count is decreased after the changes are committed or decommitted.

Deactivate Cursor

A cursor cannot be de-activated while under commitment control. The cursor must first be removed from commitment control by the Modify Commit Block instruction.

Locks Held by the Cursor at Commit/Decommit Time

After a Commit or Decommit instruction is executed, all locks held by cursors under commitment control are released.

Position of Cursor After Commit

After a Decommit instruction is executed, the position of the cursors under commitment control is not changed.

Position of Cursor After Decommit

After a Decommit instruction is executed, all cursors under commitment control are repositioned as follows:

- If the cursor was under commitment control when the commit cycle started, the position of the cursor is reset to the position that existed when the commit cycle started.
- If the cursor was placed under commitment control after the commit cycle started, the position of the cursor is reset to the position that existed when the cursor was placed under commitment control.

It is possible to remove a cursor from commitment control and again place it under commitment control during one commit cycle. In this case, the cursor is reset to the position that existed the last time the cursor was placed under commitment control.

Cursors that were under commitment control some time during the commit cycle but are no longer under commitment control at the time of decommit, are not repositioned.

Data Space Support

Data Space In-use Count

If a cursor is removed from commitment control while there are uncommitted changes to data spaces under the cursor, the data spaces are placed in-use (data base in-use count is increased) by commit. The in-use count is decreased after a Commit or Decommit instruction is executed.

Inserted Data Space Entries

If an uncommitted insert is decommitted, the result is a deleted entry in the data space. This is necessary because other committed entries may have been inserted into the data space since the uncommitted insert was made.

Data Space Entry Locks

Under commitment control, data space entry locks are held on all changed entries until a Commit or Decommit instruction is executed. Inserted entries are locked as part of the insert operation. For data space updates or deletes, the lock is not released as part of the update or delete operations. Also locks may be held on entries that are retrieved for update but are not updated allowing the process to retrieve the entry for the second time, ensuring that the entry is unchanged from the prior retrieval within the same commit cycle.

Data Space Index Support

Unique Data Space Indexes

If a key is removed from a data space index containing unique keys, the removed key is reserved until the change is committed or decommitted. Therefore, no process, including the same process using a different commit block, can add that key to the data space index.

Concurrent Data Space Index Build

If a data space index that enforces unique keys is built or rebuilt, some special considerations for commit are needed. If there are outstanding, uncommitted changes to data spaces under the data space index, a commit key index must be built. When building the commit key index, it may be discovered that if the uncommitted changes were decommitted, duplicate keys would have to be inserted into the data space index. Therefore, the data space index build is terminated and an exception is signaled.

DATA AREAS

Attached Commit Block Table

The attached commit block table, shown in Figure 1-1, provides a method for locating all of the commit blocks that are attached to a process. An entry is added to the attached commit block table when a commit block is attached to a process and the entry is removed when the commit block is detached.

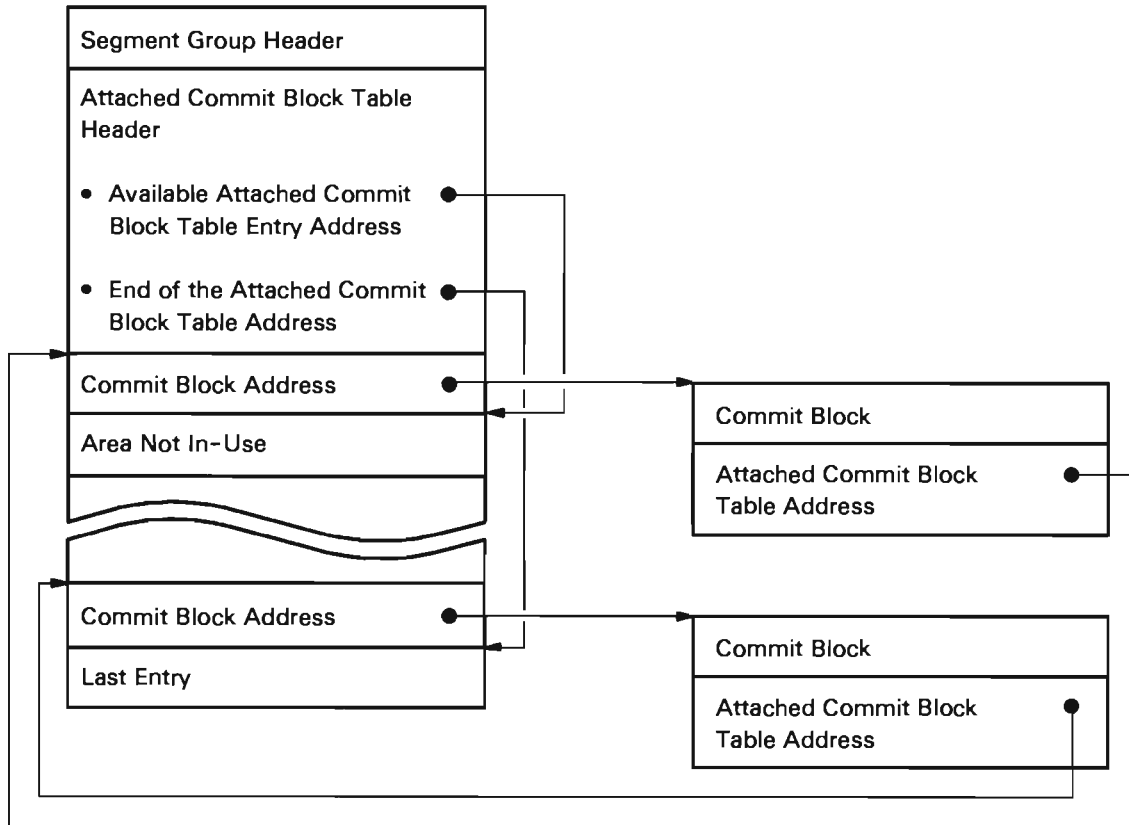


Figure 1-1. Attached Commit Block Table

Commit Block

The commit block, shown in Figure 1-2, is a permanent object that serves as the structure to control commit/decommit within a process. A commit block is associated and disassociated with a process by a Modify Commit Block instruction. Only one commit block can be attached to a process at a time.

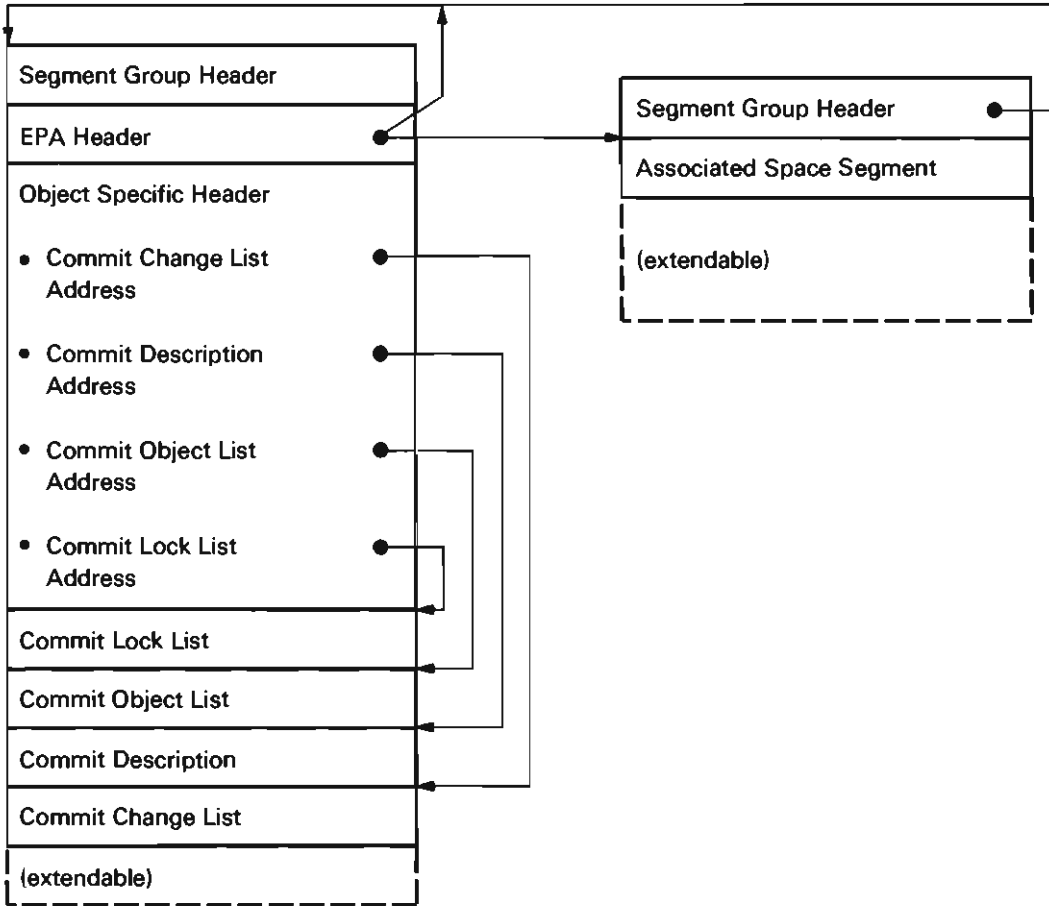


Figure 1-2. Commit Block

Commit Key Index

The commit key index is a machine index that resides in a temporary segment group. There are four types of entries possible in the commit key index. Three of the entry types (reserved, hidden, and apparent) have the same format. The fourth (start commit) has a unique format.

The format for reserved, hidden, and apparent entries is as follows:

Type of Entry	Data Space Index Key	Relative Address of Data Space Entry	Attached Commit Block Table Offset
---------------	----------------------	--------------------------------------	------------------------------------

The format for the start commit entry is as follows:

Type of Entry	Journal Port	Sequence Number	Attached Commit Block Table Offset
---------------	--------------	-----------------	------------------------------------

STRUCTURE

The following is a list of the modules in commit management and the functions that each module performs. This list also shows how the module is invoked.

#COACTFN Attached Commit Block Table Function Manager

Function: Supports the functions required to manage the attached commit block table.

How Invoked: Within this component and by the data base and journal components.

#COACTSC Attached Commit Block Table Scan

Function: Supports the functions required to scan the attached commit block table.

How Invoked: Within this component and by the data base component.

#COCBIPL Verify Commit Block Pointers and Data

Function: Validates the header of a commit block and checks all pointers in the commit block for validity.

How Invoked: Within this component.

#COCCLFN Commit Change List Functions

Function: Supports the functions required to manage the commit change list.

How Invoked: Within this component and by the data base component.

#COCHECK Cursor/Commit Checks

Function: Performs preliminary checks on a specified commit block before continuing the process.

How Invoked: Within this component and by the data base component.

#COCKIFN Commit Key Index Functions

Function: Supports the functions required to manage the commit key index.

How Invoked: Within this component and by the data base component.

#COCLLFN Commit Lock List Functions

Function: Supports the functions required to manage the commit lock list.

How Invoked: Within this component and by the data base component.

#COCOLFN Commit Object List Functions

Function: Performs all functions that modify the commit object list within the commit block.

How Invoked: Within this component.

#COCOMIT Commit

Function: Implements the Commit instruction that groups changes to an object or set of objects within one process.

How Invoked: Commit instruction.

#COCRCOB Create Commit Block

Function: Implements the Create Commit Block instruction.

How Invoked: Create Commit Block instruction.

#COCUCOB Clean Up Commit Block

Function: Cleans up the commit block after a Commit or Decommit instruction is executed.

How Invoked: Within this component.

#CODCCF Decommit Common Functions

Function: Performs the common functions to decommit the changes made under control of a specified commit block.

How Invoked: Within this component.

#CODCMIT Decommit

Function: Implements the Decommit instruction that withdraws the changes previously made under the control of a specified commit block.

How Invoked: Decommit instruction.

#CODCOEH Damage Commit Block Exception Handler

Function: Handles the recovery from an exception caused by damage to a specified commit block.

How Invoked: Within this component.

#CODSCOB Destroy Commit Block

Function: Implements the Destroy Commit Block instruction that destroys a specified commit block.

How Invoked: Destroy Commit Block instruction.

#CODUMPT Dump Task to VMC Log

Function: Dumps the process to the VMC log.

How Invoked: Within this component.

#COFORCE Force Commit Block

Function: Writes the commit block to auxiliary storage and optionally informs journal management that the commit block was rewritten.

How Invoked: Within this component and by the journal component.

#COINIT1 IMPL Recovery

Function: Performs the IMPL time recovery and initialization functions for the commit component.

How Invoked: #CFRMAST (IMPL recovery controller).

#COINIT2 IMPL Initialization

Function: Clears the attached commit block table after an IMPL object recovery function is complete.

How Invoked: #CFRMAST (IMPL recovery controller).

#COJORDE Read Journal for Commit

Function: Reads the journal addressed by the journal port in the EPA header of the specified commit block.

How Invoked: Within this component and by the data base component.

#COJOSYN Synchronize Commit Block with Journal

Function: Synchronizes the commit block with the journal during IMPL recovery.

How Invoked: Journal component.

#COULKEH Unlock Exception Handler

Function: If locks are held, this module performs the unlocking procedure.

How Invoked: Within this component.

#COMACOB Materialize Commit Block

Function: Implements the Materialize Commit Block Attributes instruction that materializes either the commit block creation template with current values, or the status of the commit block.

How Invoked: Materialize Commit Block Attributes instruction.

#COMOOBJ Add/Remove Objects from Commitment Control

Function: Performs the add objects to and remove objects from commitment control functions that support the Modify Commit Block Control instruction.

How Invoked: Within this component.

#CORELEH Release Exception Handler

Function: Exception handler to release seized objects.

How Invoked: Within this component.

#CORPCUR Reposition Cursor

Function: If the cursor is activated under commitment control and has not been repositioned, all locks that the cursor holds are placed on the commit block, then the cursor is repositioned.

How Invoked: Within this component.

#COTERM Process Termination Exit

Function: Cleans up the commit block attached to the process during VMC process termination.

How Invoked: Another VMC component.






Data Base Management

INTRODUCTION

A data base is a collection of user information stored in one or more objects called data spaces. Data base management provides the functions that allow a user to store, manage, and operate on these objects. Data base management provides:

- Late bound views of data
- Views of data independent of internal storage format
- Multiple views of the same data
- Security of data
- Integrity of managed data

Data base management supports the following System/38 instructions:

- 
- Activate Cursor
 - Copy Data Space Entries
 - Create Cursor
 - Create Data Space
 - Create Data Space Index
 - Data Base Maintenance
 - De-activate Cursor
 - Delete Data Space Entry
 - Destroy Cursor
 - Destroy Data Space

- Destroy Data Space Index
- Ensure Data Space Entries
- Estimate Data Space Index Key Range
- Insert Data Space
- Insert Sequential Data Space Entries
- Materialize Cursor Attributes
- Materialize Data Space Attributes
- Materialize Data Space Index Attributes
- Modify Data Space Index Attributes
- Release Data Space Entries
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entries
- Set Cursor
- Update Data Space Entry

Some of the internal functions supported by data base management are as follows:

- Build data space index
- Build composite key
- Build logical key
- Generate field mapping code
- Invalidate data space index
- Modify in-use table

- Remove index addressability from data space header
- Remove addressability to data space entry from locked entry queue
- Verify mapping template
- Conversion/mapping exception handler
- Unlock data space entry
- Bring data pages
- Detect pseudoduplicate keys
- Clone data segments
- Re-validate deleted entry count
- Log delayed key maintenance
- Force locked entry
- Force data space
- Force all indexes
- Force recently inserted entries
- Force recently modified entries
- Handle entry spanning segment identification (SID) group boundary
- Discard all data space index directory blocks
- Merge mini-indexes
- Clear data SIDs
- Derive ordinal number
- Derive data space entry virtual address
- Perform appropriate journaling of data base modifications

Data Sharing

Data base management provides for the sharing of data among concurrent processes. The following paragraphs describe the methods used to share data and the procedures used to ensure the integrity of the shared data.

Cursors

The cursor is the only object that makes data accessible to a process. The data space and data space index do not contain any process related information. The process dependent information is kept in the active cursor, which is usable only by the activating process. However, multiple processes can each simultaneously have cursors over the same data spaces and data space indexes allowing the processes to share the data. Every instruction that accesses data in a data space obtains the data by using a cursor; data is not accessible through any other mechanism.

The cursor indicates the entry currently addressed for retrieval and the entries locked for update. It contains the information needed to map the internal format of the entries into the format desired by the process. The following types of mapping are supported:

- Field rearrangement
- Skipped fields
- Numeric field conversion to other numeric types
- Character field truncation and padding
- Derived field
- Joined records
- Record selection

These locks provide the following functions:

- Prevent the destruction of the object while in use
- Prevent a cursor from being used by more than one process at any one time
- Prevent another process from obtaining a lock exclusive no read lock on any data base objects shared by a cursor

These locks are removed when the cursor is de-activated.

If a cursor is active and a set cursor operation for an update is performed, the data space is implicitly locked with a lock shared update lock unless the lock applied by the Activate Cursor instruction is adequate. This implicit lock on the data space is not removed until all the locked entries in that data space are updated, deleted, or released. Data base management will place only one implicit lock shared update lock on a data space from a given cursor.

Locked Entries

Between the time an entry is located by a set cursor operation and the time it is modified (updated or deleted), the entry must be protected from modification by other processes. Data base management locks the entry during a set cursor operation if an update or deletion of the entry is specified. The lock prevents other processes from updating or deleting the entry. The locking process can only hold one lock per data space entry. Update and delete operations subsequently unlock the entry. An entry can be unlocked without change by using a Release Data Space Entries instruction or by de-activating the cursor holding the lock.

A set cursor operation applies an implicit lock to each data updated. The list of hold records associated with the locked data space entries are chained from the cursor by head and tail pointers as shown in Figure 2-1. The list is organized as both a last-in-first-out queue and a first-in-first-out queue. An entry can be added to the head of the queue or to the tail of the queue. When an entry is removed from the queue, the entry lock is removed from the head of the queue. The entry lock can be removed from the head or tail of the queue by using the Release Data Space Entries instruction. If the cursor is under commitment control, the entry locks must be transferred to the commit block.

The last four bytes of the hold records contain the following information:

- Data space number (1 byte)
- Flag information (1 byte)
- Lock chain link (2 bytes)

The virtual address in the hold record addresses the first byte of the locked data space entry. Because the entries are locked with a lock exclusive allow read lock, other users can only retrieve the locked entries.

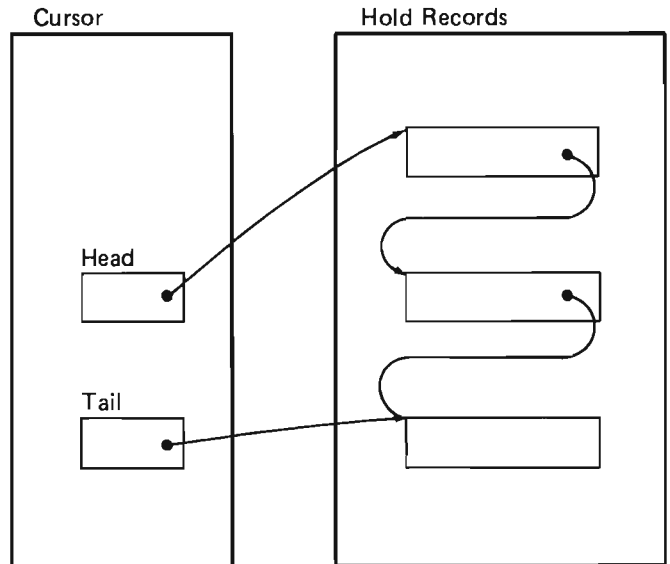


Figure 2-1. Hold Records List

In Use

When a cursor is activated, the data spaces and data space index referenced by the cursor are marked as in use. This is accomplished by incrementing the use-count associated with both the data spaces and the data space index. A data space index marked as in use (a nonzero use-count) cannot be destroyed or explicitly invalidated. When the cursor is de-activated, the use-counts for these objects are decremented. A nonzero use-count prevents any process from destroying the data space index.

For recovery reasons, the use-count for both a data space and a data space index is kept in the data base in-use table. These counts are reset during initial program load (IPL). One routine (#DBXMUSE) performs all operations on the table. The in-use table is initialized during each IPL (after recovery operations are complete). The table consists of rows of object identifications. It is seized and released every time it is referenced and is seized and released only within #DBXMUSE.

An entry with a use-count and a retain status equal to 0 identifies a free slot in the table. When a data space or a data space index is to be operated on, an entry in the in-use table that specifies that object is found and the use-count is incremented. If the object is not found, the first free slot is assigned to the object and the use-count is incremented. If an entry has been added to or deleted from the in-use table, the table is saved on auxiliary storage.

When an object is removed from the in-use state, the table is seized and then searched. When the object is located, its use-count is decremented.

A data space index entry is placed in the in-use table when the index is in the process of being created, rebuilt, loaded, or used as an access path. An entry is removed when the operation is completed. This type of entry is used at recovery time to determine the recovery actions for a data space index that was being operated on when a system failure occurred.

Note: The in-use table is saved (forced) on auxiliary storage each time an entry is inserted into or deleted from the table. It is not saved each time the use-count is changed because the recovery operations proceed regardless of how many processes were simultaneously using the object.

Load/Dump and Suspend

The basic concept in data base load/dump operation is the network. A network is defined as a grouping of data spaces and data space indexes such that if a data space index is a part of the network, then every data space that is referenced by that data space index is also a part of the network. (However, if a data space is a part of the network, then every data space index that is referenced by that data space need not be a part of the network.) The smallest possible network is a single data space. The largest possible network is the group of all data spaces and data space indexes in the system.

Figure 2-2 shows an example of a network configuration. The following is a list of the valid data base networks that can exist in that figure:

- Data space index, data spaces 1, 2, and 3
- Data space index, data spaces 1 and 2 (example shown)
- Data space 1
- Data space 2
- Data space 3
- Data spaces 1 and 2
- Data spaces 1 and 3
- Data spaces 2 and 3
- Data spaces 1, 2, and 3

Load/dump will only dump or restore complete networks, although the restored network can be a subset of a dumped network. When loading a network, all data spaces must be loaded before the indexes (if any) are loaded.

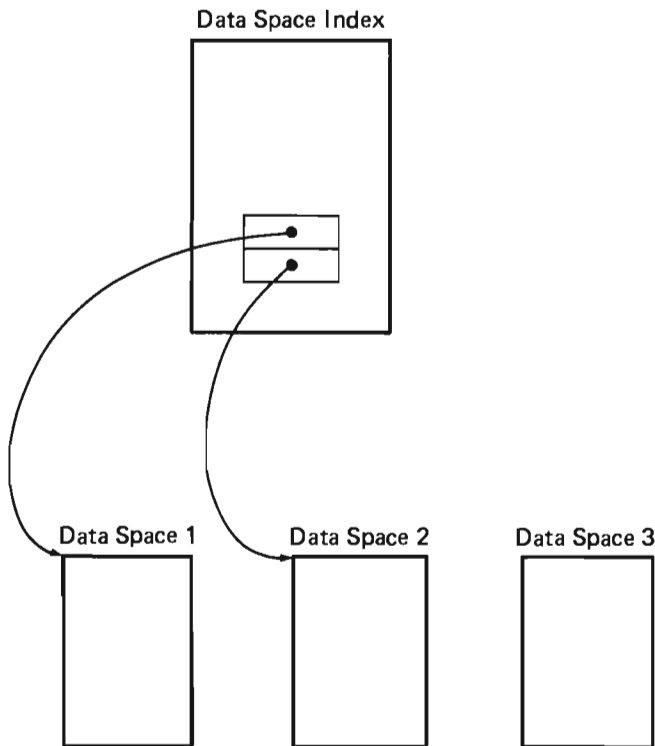


Figure 2-2. Networks

Generally, load/dump performs seizing on a network basis. Load/dump is responsible for seizing all objects including the data space indexes that are referenced by data space that is being overlaid by a load operation but are not themselves being loaded. The invoking load/dump routines will not load a data space if its field description table is not identical to the field description table of the data space being overlaid. This ensures that indexes can be rebuilt validly.

Three data base functions are provided for the exclusive use of the load/dump function:

- Fix data space header (#DBXFDSH)
- Fix data space index header (#DBXFIXH)
- Clean up at end of network (#DBXRINX)

These routines provide most of the object specific actions required during loading of a data base object. #DBXFDSH is invoked by load/dump after each data space is loaded. This routine updates pointers in the header and then ensures that the old network is deleted and the new one is correctly loaded. If this is a create and load operation, then the load/dump flag is set to on in each data space index block. When the data space indexes are loaded, this flag indicates that the index pointer is from the address space of the machine that performed the dump rather than the address space of the machine performing the load. During network cleanup, this flag indicates that the data space index block is invalid (because no index was loaded) and should be deleted.

If this is a replacement type load, then each data space index block from the overlaid header is also copied into the loaded header. Because the data space index blocks from the overlaid header are copied, each is checked to determine if the index it references is valid. If the index reference is valid, then the index is invalidated (without signaling an event) and the index invalidated flag is set to on. If the data space index block is not subsequently deleted because a new version of the index was not loaded, the invalidated flag causes an index invalidated event during network cleanup.

#DBXFIXH is invoked by load/dump after each data space index is loaded. In addition to updating the internal pointers and pointers to the data spaces, this routine examines each data space and performs the following operations:

- If there is a data space index block in the data space that points to the address currently occupied by the index and the load/dump flag in this block is off, #DBXFIXH sets the load/dump flag to on and sets the pointer to zero. This causes the network cleanup routine to delete the block.
- If there is a data space index block in the data space that points to the address of the index at the time the index was dumped, and the load/dump flag in this block is set to on, #DBXFIXH updates the pointer to the current address and sets the load/dump flag to off.

#DBXRINX is invoked when the loading of the network is complete (a data space with no indexes is considered a network). This routine searches the data space index blocks for entries with the load/dump flag set to on and deletes the entries. This routine also searches for data space index blocks with the invalidated flag set to on and signals an event for each index (turning the flag off at the same time). This event can be used to indicate that the index requires rebuilding.

Data Base Management Recovery and IPL

Recovery after an abnormal machine termination involves restoration or destruction of objects. The data base recovery phase manipulates the following objects and internal structures:

- Data spaces
- Data space indexes
- Cursors
- In-use table

Most data base recovery operations are performed during IPL. The operations performed are determined by the contents of the in-use table. The data base objects recovered as a result of references contained in the in-use table are identified in the object recovery list.

The recovery operations performed during IPL affect all data base objects except active cursors. Active cursors are recovered during a subsequent attempt to activate the cursors.

The in-use table (DB#DSIU) contains entries that identify the data base objects that were being used when the machine termination occurred. The objects identified in the table include data spaces and data space indexes. After all the entries in the in-use table have been processed for recovery, the in-use table is reset (all entries removed) in preparation for the next IPL.

Data Space Recovery

The following actions are performed for each entry contained in the in-use table that references a data space:

1. All recently inserted entries are read, starting with the most recently inserted entry on auxiliary storage and progressing until an entry is detected for which the insert operation was not completed.
2. The entry count and force count within the data space header are updated to reflect the ordinal number assigned to the final recoverable entry.
3. Information identifying the data space and its recovery status are appended to the object recovery list.
4. Any pending recovery action is performed.
5. All data space indexes affected by the contents of the recovered data space are invalidated unless internal flags indicate that the binary tree of the index was not recently changed without being forced to auxiliary storage. Information identifying the invalidated indexes is then appended to the object recovery list.

Data Space Index Recovery

Data space indexes are referenced in the in-use table during the following:

- Creation of the data space index
- Rebuild of the data space index
- Use of the index as an access path
- Loading an index

Indexes that were in the process of being created when a termination occurred are destroyed during the recovery phase.

Indexes that were being rebuilt when a termination occurred are invalidated and identified on the object recovery list during the recovery phase. Indexes whose binary trees were modified but not written to auxiliary storage are invalidated and identified on the object recovery list. Indexes that were being loaded are flagged as damaged.

Cursor Recovery

Cursor recovery, if required, is performed during subsequent attempts to activate the cursor. The current process identification and the IPL number are examined to determine if recovery for the cursor is required. If the process identification in the cursor does not match the current process identification, and the IPL number in the cursor does not match the current IPL number, then the cursor was activated at the time the machine failure occurred. If this is the case, the appropriate fields are updated and the cursor is reactivated.

DATA AREAS

The machine interface objects supported by data base management are:

- Data space
- Data space index
- Cursor

The major data area created and used internally by data base management is the data base in-use table.

Data Space

A data space (shown in Figure 2-3) consists of three or more segment groups. The first segment group contains:

- The segment group header
- The encapsulated program architecture (EPA) header
- The object specific header containing:
 - Length of index chain.
 - The maximum number of data segment groups that can be allocated.
 - Number of data segment groups actually allocated (usually one).
 - Pointers to the data segment groups. These pointers address the last byte allocated in the segment group. The address of the first byte of data in the segment group is obtained by putting 0's in the right three bytes and adding hex 20 (the length of the segment group header).
- A variable-length chain identifying the data space indexes over the data space.

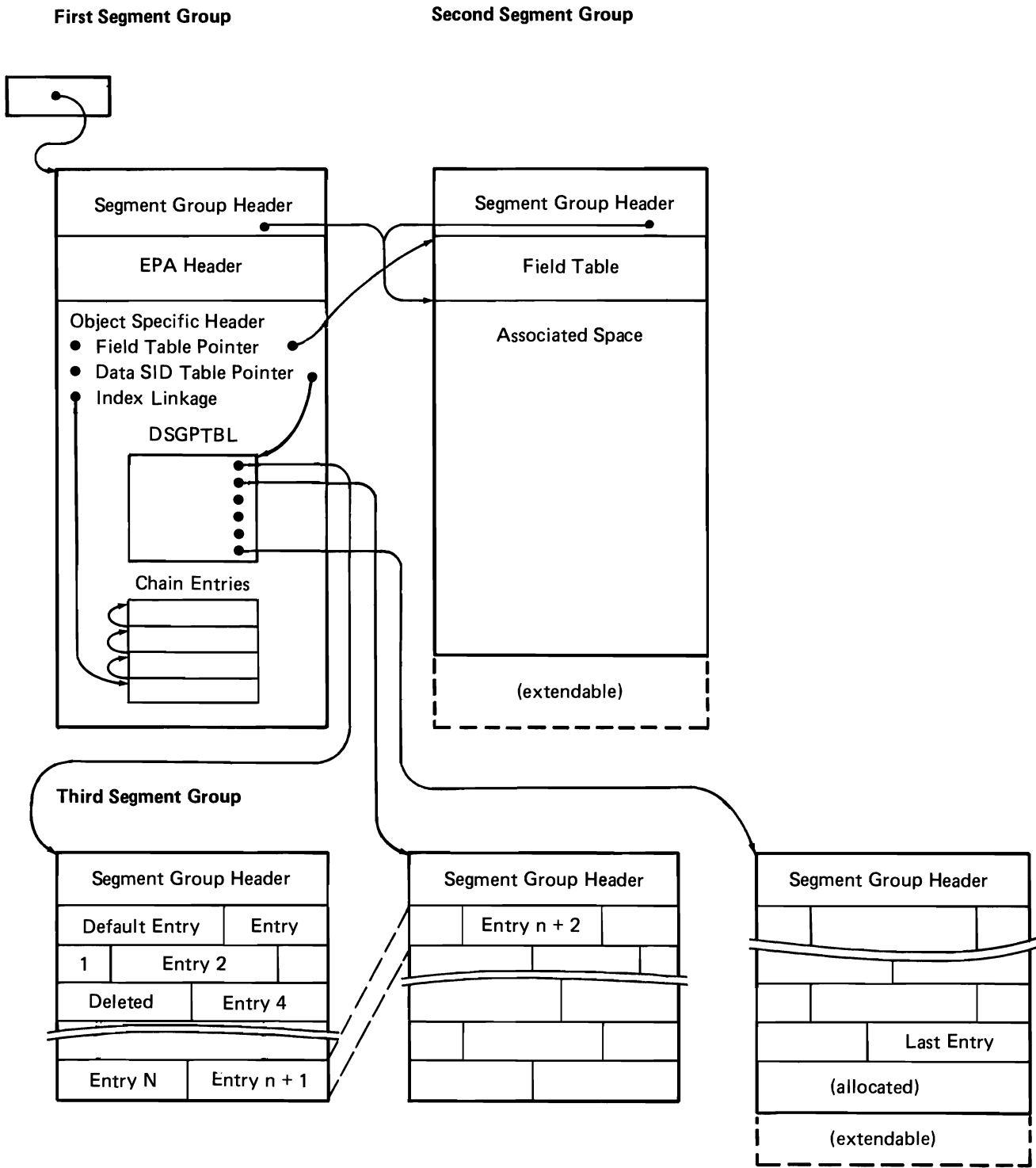


Figure 2-3. Data Space

The entries in the variable-length chain contain:

- The address of the next entry on the chain.
- The address of the data space index.
- Code generated by a Create Data Space Index instruction used to compare a data space entry to a revised entry to determine if any key or selection fields have been changed as a result of an Update Data Space Entry instruction. If the fields have not changed, the index is not affected by the update and does not need modification.

When a data space index is created, a chain entry for that data space index is added to the end of the data space header for every data space covered by the data space index. Linkage to a new entry is added to the front of the chain. When a data space index is destroyed, the entry for that data space index is deleted from the chain and all subsequent chain entries are moved up in the segment group and the linkages adjusted. The blocks of the chain within a data space header are linked in reverse order of their creation. Every chain entry is potentially a different size due to the variable quantity of generated code residing in the block.

The second segment group contains:

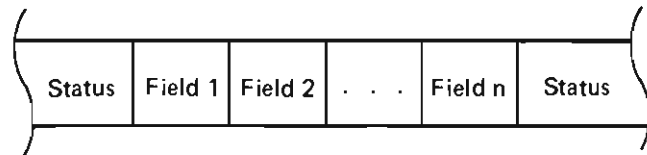
- A segment group header
- A table defining the data space entry fields (field table)
- The associated space

The field table is used by create cursor, create data space index, and materialize data space attributes functions. The associated space (if present) follows the field table and is extendable.

The third and subsequent segment groups contain:

- A segment group header
- Data space entries
- Status information concerning each entry

The data space entries are stored in the data segment groups specified by the segment group table contained in the object specific header. All entries are strung together end-to-end, and are separated by a status byte. A typical entry would be:



where:

- Status contains a flag that the entry:
 - Is valid
 - Is deleted
 - Crosses a segment group boundary
- Field 1 through Field n are ordered as defined in the field table contained in the second segment group.

The segment groups can be viewed as one contiguous addressing space (minus the segment group headers). Entries can span segment group boundaries. The initial entry in the first data segment group is a default entry and is used to supply values for fields when inserting entries and updating deleted entries when a value is not supplied by the user. The procedure for calculating the address of an entry is shown in Figure 2-4.

Data Space Index

The data space index is made up of a primary segment identification (SID) and 67 optional segment groups as shown in Figure 2-5 and is addressed by a system pointer. The primary SID group contains:

- The segment group header.
- An EPA header.
- An object specific header containing:
 - Values and attributes
 - Alternate collating sequence table (if alternate collating is specified)
 - Translate table
 - Intermediate mapping table
 - Non-user exit selection table
 - Key specifications
 - Selection specification
- A machine index.

The first optional SID group contains:

- The segment group header.
- An associated space.

The second optional SID group contains:

- The segment group header.
- The selection routine if a select/omit data space index.

The third optional SID group contains:

- Segment group header.
- A log containing the changes made to all the data spaces under the data space index since it was last maintained. The remaining 63 segment groups contain entries that constitute the expanded binary tree.

The fourth optional SID group is present only for data space indexes that require unique keys and the data space index has had a key deleted by a process running under commitment control. This SID group contains:

- Segment group header
- Commit key index header
- Machine index containing data space index keys that were deleted but are reserved to prevent duplicate key conflicts from occurring during decommit

Alternate Collating Sequence Table

The alternate collating sequence table in the object specific header is a 256-byte translation table that is present only if alternate collating was specified when the data space index was created. Machine indexes order their keys according to their binary values. If the user wants a character key field or a binary field to be ordered differently than the normal binary (EBCDIC) ordering, that field in the key can be translated to an alternate collating sequence.

Translate Tables

Translate tables are used with intermediate mapping to replace the actual characters in the data space with a value defined in the translate table. A user can specify an array of tables and the mapping specifications control which table is used from the array.

Intermediate Mapping

If intermediate mapping is specified, the intermediate mapping table is made up of one IKEY entry for each DS under the DSI. The scalar portion of the object specific header contains the address of the table. Each IKEY entry contains:

- The address of the mapping template.
- The address of the generated mapping code.
- The address of the field description table for the intermediate buffer
- The address and length of the data literal area.
- Related flags and statistics.

Nonuser Exit Selection

If nonuser exit selection is specified, the selection table is made up of one SEL entry for each DS under the DSI. The scaler portion of the object specific header contains the address of the table. Each SEL entry contains:

- The address of the selection template.
- The address of the generated selection code.
- The address and length of the data literal area.
- Related flags and statistics.

For user exit selection, the select/omit specification area is described in the select/omit section (User Exit Selection).

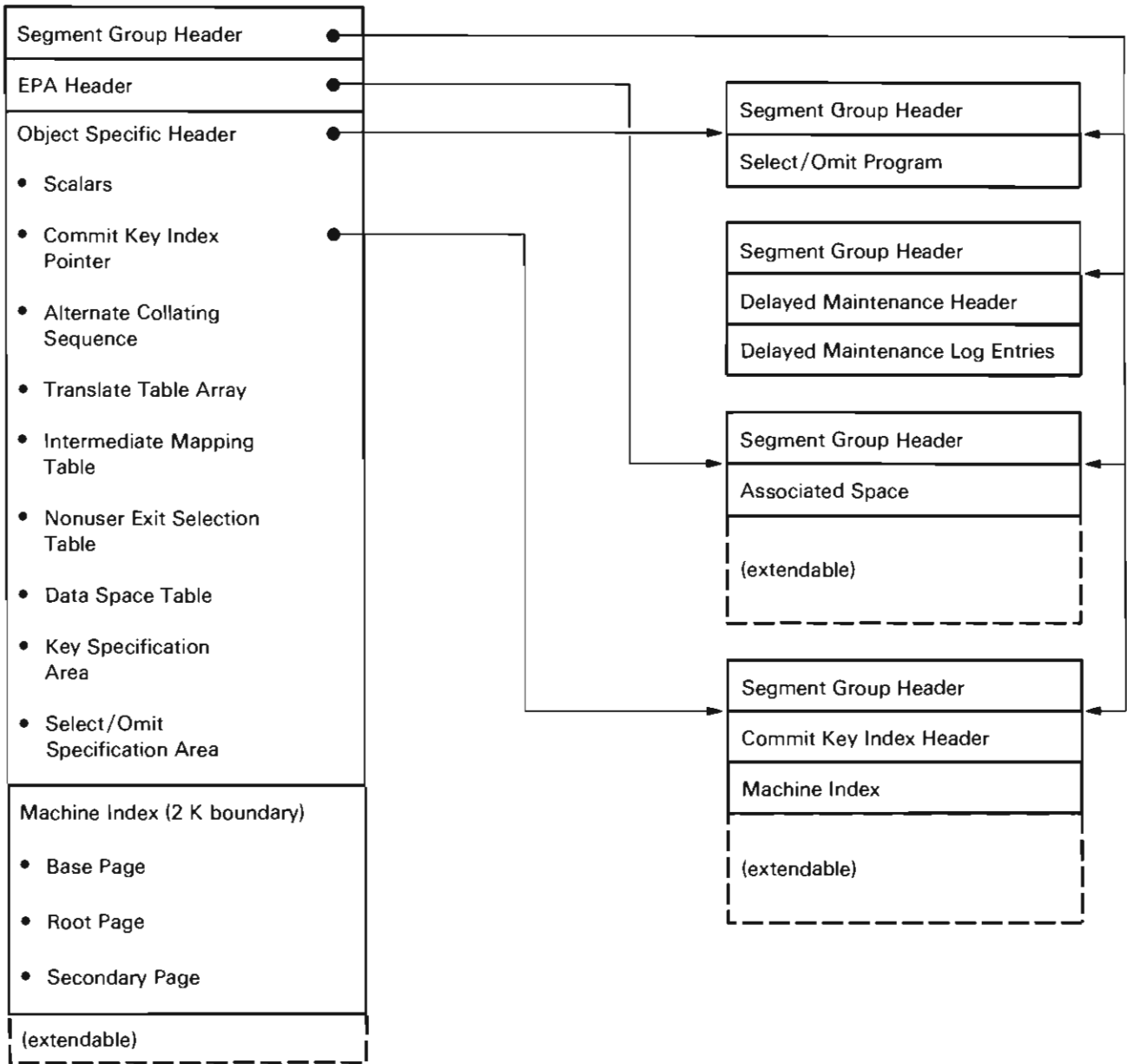


Figure 2-5. Data Space Index

Key Specification Area

The key specification area is composed of a number of tables as shown in Figure 2-6. The first data space key (DKEY) table contains a row (entry) for each data space covered by the data space index. Each row contains:

- The address of the data space
- The number of rows in the key field description table for that data space
- A pointer to the key field description table (DKYT) for that data space

The number of rows (number of data spaces covered by the data space index) in the DKEY table (DB#DKEY) and its address are contained in the object specific header. The row number in the table is the data space number required for keyed-cursor and insert-data-space-entry operations.

There is a DKYT for each row in the DKEY table. The DKYT defines the fields that form the key. Each row in the DKYT defines the attributes of one field in the key. These attributes are:

- The offset into and the location of the field in the data space entry
- The length of the field (or value of the fork character if specified)
- Attributes of the field, such as:
 - Ascending/descending sequence
 - Absolute value
 - Alternate collating sequence
 - Fork character
 - Zone/digit force

DB#DKEY

Row 1	Pointer to Data Space	Number of Keys	Number of Binary Tree References	Highest Ordinal Number	Key Length	Pointer to DKYT
Row 2	Pointer to Data Space	Number of Keys	Number of Binary Tree References	Highest Ordinal Number	Key Length	Pointer to DKYT

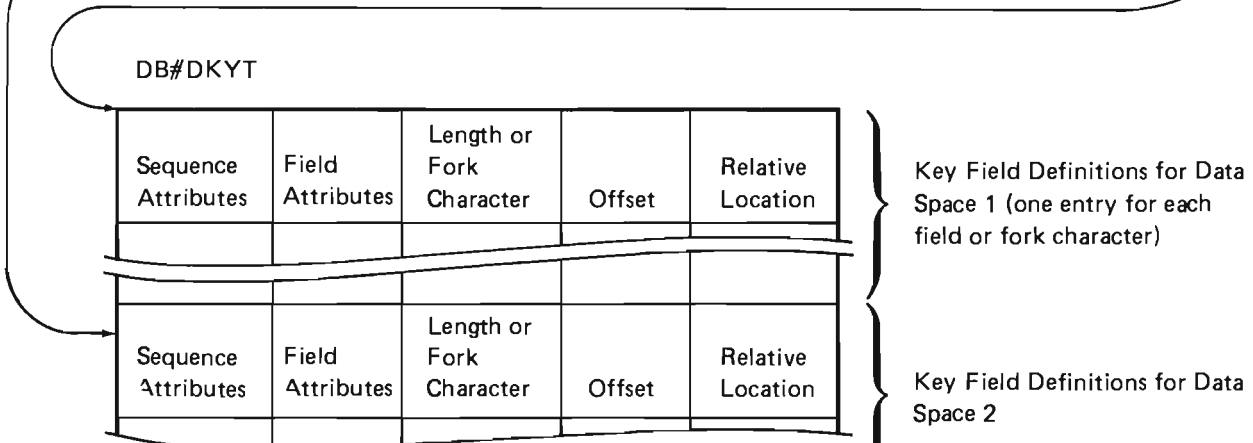
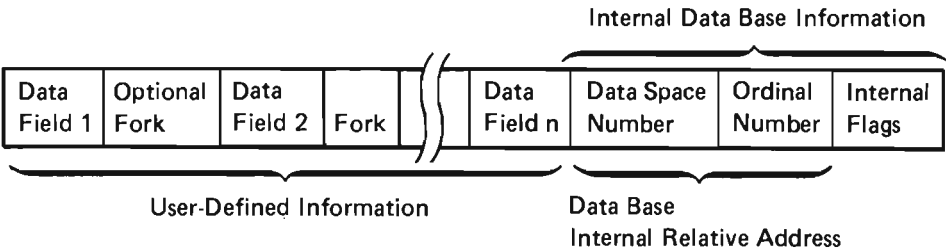


Figure 2-6. Key Specifications

The order of the rows in the DKYT for a data space defines the order of the fields in the composite key, and is used by a table driven routine that builds composite keys from either the data space entry or the physical image derived from a user provided logical key.

Data base management uses machine indexes to store and order keys in a data space index. An index control block (IXCB) is used to communicate with machine index management. Each time an index is to be used, it is seized and the IXCB is loaded with the values needed to operate on the index. When the operation is completed, the index is released.

Because machine indexes handle every entry as a bit string, and order them in ascending order, data base management must perform several operations on the key fields to obtain the desired ordering specified by the user. A data base key is made up of one or more data space entry fields, and optional single characters called fork characters. The format of the key is:



The user defines how the key is built to generate the desired order in the index when the data space index is created. After each key field has been built, data base management takes the bit string of each key field or fork characters, builds the internal information, and inserts or removes the key into or from the machine index portion of the data space index. Routine #DBXBLKY performs the key building function. Following are the operations performed by #DBXBLKY to build a key for each field in an entry and to insert or delete the key from the machine index:

- Force zone or digit: The selected 4 bits of each byte within the key field are set to 0.
- Alternate collating: Each byte of the key fields is translated and replaced with the appropriate character from the alternate collating table.
- Numeric ordering: A key field can be ordered in one of the following ways:
 - Unsigned bit string
 - Algebraic value (signed)
 - Absolute numeric value

Because the signs of binary, zoned, float, and packed fields are located at different locations within a field, using an unsigned bit string ordering results in the following sequence:

- Binary (sign is the leftmost bit): 0 to positive infinity followed by negative infinity to -1.
 - Packed (sign in numeric field of rightmost byte): The order depends on valid sign value. Generally, the order is 0, -0, 1, -1, . . . , n, -n.
 - Zone (sign in zone field rightmost byte): The order depends on valid sign values. Generally the order is 10 positive numbers followed by the corresponding 10 negative numbers.
 - Float (sign on left): The ordering is 0 to infinity and then the smallest magnitude negative number to infinity.
- Order: The order sequence indicator specifies that the key field is to be complemented to produce descending sequence.

Figure 2-7 shows the key conversion rules used by #DBXBLKY when building a key. The figure also shows an example of the conversion performed.

The internal information that is added to the user key to build the index key consists of the data base relative address and some internal flags. The data base relative address consists of the following:

- The adjusted number of the data space in which the entry is contained (0 origin)
- The ordinal number of the entry

Value	Field Type	Sign	Rule	Example	
				Before	After
Absolute	Binary	Positive	No change	0003	0003
		Negative	Take twos complement	FFFD	0003
	Zoned Decimal	Positive	Force zone fields to 0	F0F3	0003
		Negative		F0D3	0003
	Packed Decimal	Positive	Sign bits to hex F	003F	003F
		Negative		003D	003F
	Floating Point	Positive	No change	40400000	40400000
		Negative	Force sign bits to binary 0	C0400000	40400000
Algebraic	Binary	Positive	Force sign bit to negative	0003	8003
		Negative	Force sign bit to positive	FFFD	7FFD
	Zoned Decimal	Positive	Force zone fields to hex F	F0F3	F0F3
		Negative	Force zone fields to hex F, then take ones complement	F0D3	0F0C
	Packed Decimal	Positive	Force sign bits to hex F and move to the left 4 bits	003F	F003
		Negative	Force sign bits to hex F, move to left 4 bits, then take ones complement	003D	0FFC
	Floating Point	Positive	Force sign bit to binary 1	40400000	C0400000
		Negative	Complement entire field	C0400000	3FBFFFFF

Figure 2-7. Key Conversions

The adjusted data space number within the composite key designates the relative position of the data space in the data space list starting from position zero (contained in the template for the Create Data Space Index instruction). The encoded ordinal number designates the position of the entry in the data space (first user-supplied entry has ordinal number of one). For an index with a last-in-first-out ordering, the ordinal number is complemented to place the high ordinal numbers before the lower ordinal numbers.

The appended relative address field ensures unique keys within the machine index, and the encoded information is used when the key is subsequently retrieved to locate the entry in the data space. The relative address also allows relocation of data space indexes from system to system without rebuilding the index.

Fork characters allow the user to define the order of keys of different lengths. When a user creates an index over two or more data spaces or the same data space multiple times, the designated key length for each data space can vary in length. Because the key is made up of the user key followed by the machine-supplied database relative address, the order of the keys in the index is not user controllable without fork characters. Figure 2-8 shows two data spaces with variable-length keys. The figure also shows examples of the key ordering, one without fork characters, and one using fork characters.

Note: The example orderings are field aligned for ease of use.

Data Space 1:

Entry Number		Key Field Char (2)
01	(AB)	C1 C2
02	(CD)	C3 C4
03	(EF)	C5 C6
04	(D4)	C4 F4
05	(34)	F3 F4

Data Space 2:

Entry Number		Key Fields Char (2)	Bin (15)
01	(AB)	C1 C2	00 00
02	(AB)	C1 C2	00 01
03	(34)	F3 F4	03 00
04	(34)	F3 F4	00 04
05	(34)	F3 F4	05 06

The following is the resulting index without using fork characters:

User Key	Data Space Number	Ordinal Number
C1 C2	00	00 01
C1 C2 00 00	01	00 01
C1 C2 00 01	01	00 02
C3 C4	00	00 02
C4 F4	00	00 04
C5 C6	00	00 03
F3 F4	00	00 05
F3 F4 00 04	01	00 04
F3 F4 03 00	01	00 03
F3 F4 05 06	01	00 05

The following is the resulting index using fork characters FF for data space 1 and 00 for data space 2:

User Key				Data Space Number	Ordinal Number
		Fork Character			
C1 C2	00	00	00	01	00 01
C1 C2	00	00	01	01	00 02
C1 C2	FF			00	00 01
C3 C4	FF			00	00 02
C4 F4	FF			00	00 04
C5 C6	FF			00	00 03
F3 F4	00	00	04	01	00 04
F3 F4	00	03	00	01	00 03
F3 F4	00	05	06	01	00 05
F3 F4	FF			00	00 05

Figure 2-8. Example Key Ordering

Select/Omit Specification Area

The select/omit specification area of the header is only present if select/omit was specified when the data space index was created. The tables in the select/omit specification area are used to map fields to a selection buffer for passing to the user exit program provided by the machine interface. This routine determines whether or not a data space entry is to be addressed by a data space index. The select/omit specification area of the header is made up of a number of tables (a data space select/omit table and a variable number of data space select/omit field description tables) as shown in Figure 2-9. The data space select/omit table has one row for each data space covered by the data space index for which there is select/omit specifications. The row contains values for:

- The number of entries not addressed by the index (the number rejected)
- The number of fields to be passed to the selection routine
- The length of the selection buffer
- A pointer to the generated mapping code
- The length of the selection mapping code

Following the data space select/omit table are some number of areas, each containing:

- Generated code for mapping fields from the data space entry to the selection buffer
- The data space select/omit field description table defining the fields to be mapped from the data space entry to the selection buffer

There is an area containing generated code and a data space select/omit field description table for every data space select/omit table entry that specifies one or more fields to be passed to the user exit program.

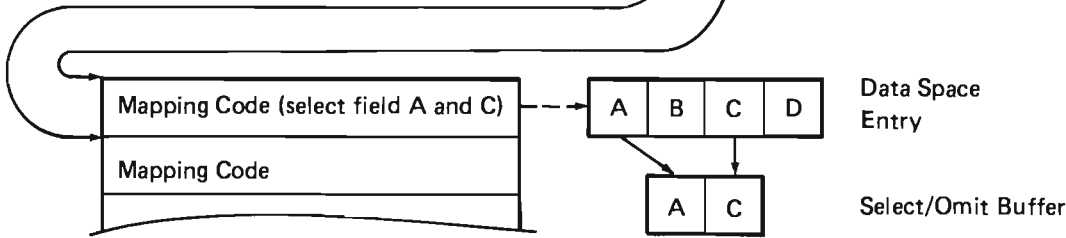
The generated mapping code is used to fill the selection buffer before the buffer is passed to the user exit program. The mapping code is used by update and insert functions when a select/omit field is modified. The mapping code is also used by the index build function.

The data space select/omit field description table is used by the materialize function to materialize the data space index template. The pointer to the generated code and data space select/omit field description table in the data space select/omit table row for a data space is used to branch to the generated code. The length of the generated code is also in the data space select/omit table row and is used to find the data space select/omit field description table that follows the generated code for that data space.

The machine index trunk page starts at a 2 K boundary following the end of the object specific header. The initial allocation of the space for the header segment group is based on the current number of entries in the data spaces, and ensures that the machine index has at least three logical pages available, and that the area allocated for the header segment group is a multiple of 4 K bytes. Refer to *Data Space Index* in this section and to the *Machine Index Management* section in this manual for additional information concerning the machine index.

Data Space Select/Omit Table (DB#DSEL)

Number Rejected	Number of Fields	Buffer Length	Pointer to Mapping Code	Code Length	} Entry for Data Space 1
Number Rejected	Number of Fields	Buffer Length	Pointer to Mapping Code	Code Length	
					} Entry for Data Space 2



Data Space Select/Omit Field Description Table (DB#DSLTL)

Field Location	Field Type	Field Length	} Entries for Fields in Data Space Entry 1 (one entry for each field)
Field Location	Field Type	Field Length	
Field Location	Field Type	Field Length	

Figure 2-9. Selection Specifications

User Exit Selection

The user exit program allows the user to provide data space indexes that address a subset of the entries in the data spaces. The user exit program contains the logic that determines if a data space entry is selected or omitted from a data space index.

The user exit program is called implicitly from data base instructions that perform index maintenance. These instructions are:

- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Update Data Space Entry
- Create Data Space Index
- Data Base Maintenance with Rebuild Option

The user exit program is passed a space pointer as an argument that points to a buffer that contains the following information:

- **Answer Field:** This field is set by the user exit program to direct the data base instructions that called the user exit program to insert or not insert the key into the data space index.
- **Data Space Number:** This field can be used by the user exit program as input to make a decision on the insertion of the key into the data space index. This field contains the number of the data space that contains the entry. Since an index can be over more than one data space, the user exit program uses this field to identify the correct data space.
- **Data Space Entry Fields:** These fields are used by the user exit program to make a decision on the insertion of the key into the data space index. These fields are in the order and format as defined by the data space index.

Creating a DS Index from an Existing DS Index

The user has the option of creating a DS index from an existing (source/parent) DS index rather than directly from the underlying data spaces. The user must specify a subset of the source DS index entries to be used in the new index. The key specifications provided are compared with those in the source DS index to determine if the new key specifications can be built purely from fields in the keys of the source DS index, which is faster than building the new keys from the underlying data space entries. If the source DS index contains nonuser exit selection, the selection template of the source is reproduced in the new DS index and merged, if necessary, with any newly specified selection template. The resultant selection template is then used to generate the final selection code. If any of the selection criteria relies on fields in the source DS index intermediate buffer, then the intermediate mapping templates undergo a similar operation. If the source DS index intermediate mapping template is required in the new DS index, then its corresponding translate tables must be similarly reproduced in the new DS index. Once created, the new DS index has no need to maintain any linkage to the source DS index.

When a data space index is created, the user exit program (if provided) is copied into the data space index. The user exit program is placed in the third segment group of the data space index. Only the following parts of the user exit program are copied:

- EPA header
- Instruction stream
- Static initialization code
- Program header
- Program template
- Breakpoint offset mapping table

The remaining parts of the user exit program are ignored since functions (for example: Trace instructions) that require them are not used for user exit programs. The addresses within the copied program are altered to reference the copied segment group.

Copying the user exit program into the index ensures the continued existence of the user exit program.

Cursor

The cursor, shown in Figure 2-10, is represented by two segment groups.

The first segment group contains such things as:

- The segment group header
- An EPA header
- An object specific header (cursor header) that contains things such as:
 - Attributes and status
 - Data space mapping code (DMAP)
 - Join cursor key mapping code (JMAP)
 - Data space selection code
 - Key buffer (if cursor is over an index)
 - Index control block (if cursor is over an index)

The second segment group contains:

- The segment group header
- Associated space

Segment Group Header
EPA Header
Cursor Header
Attributes and Status
Data Space Mapping Table (DMAP) Join Cursor Key Mapping Table (JMAP) Group Definition/Description Table Cursor Selection Map Data Selection Routine (one per DMAP) Group Selection Routine Translate Tables
Data Space Mapping (DMAP) Code (one for each data space in list) Generated Input Mapping Code Generated Output Intermediate Mapping Code Generated Output Mapping Code Generated Input Key Mapping Code Generated Output Key Mapping Code
Join Key Mapping (JMAP) Code (repeated for each secondary data space) Group Default Primitives Entry Group Logical Key Default Entry Generated Group Primitive Procedure Code Generated Group Intermediate Map Code
Generated Data Space Selection Code (one for each data space) Generated Cursor Intermediate Key (CRIK) Selection Generated Cursor Intermediate Key (CRIK) Mapping Code Fast Search Array for this data space Merged Fast Search Array Generated Group Selection Code IXCB for Primary Index
Key Buffer for Primary Index Look-Ahead Buffer for Primary Index
IXCB for Secondary Index (one for each secondary DS)
Key Buffer 1 for Secondary Index Key Buffer 2 for Secondary Index
Intermediate Buffer for a Join Cursor Buffer for DS Entries for a Join Cursor Copied DS Mapping Template (one for each DS) Copied DS Selection Template (one for each DS) Group Buffer Definition Template, Selection Template, Mapping Template Join Definition Template Data Literals for all Templates Segment Group Header Associated Space

Figure 2-10. Cursor

The DMAP contains a row (entry) for each data space that the cursor is over. The data spaces are in the same sequence as in the data space list supplied during creation of the cursor. If the cursor is over a data space index, the sequence in the data space list and in the index is identical. This is checked at creation time by comparing the list to the index. A subset of data spaces can be specified at creation time by supplying zeros instead of a system pointer for an entry in the data space list; this causes a zero DMAP entry to be built. The zero entry is built because the DMAP entry number is the same as the data space number used by a set cursor operation.

A row in the DMAP that identifies a data space contains the following information:

- A pointer to the data space.
- Pointers to the mapping code used to map the logical view to the actual physical form in the data space. Up to five sets of mapping code can be present:
 - Two sets are used for data mapping and are always present
 - Data to: Maps the data to the data space. Used by insert and update functions.
 - Data from: Maps the data from the data space. Used by the retrieve function.
 - Two sets are used for key mapping and are present only if the cursor is over a data space
 - Key to: Maps the logical key to the physical format. Used by the set cursor function before the composite key used as a search argument is built.
 - Key from: Maps a physical key to the logical format. Used by the materialize cursor attributes and set cursor functions to return the logical key.
 - The fifth set of mapping code is used for derived field mapping:
 - Derived Field Operations Mapping/Deriving From Data Space: Used by set cursor, retrieve data space entry, retrieve sequential data space entry.
- Direct map indicator for the data mapping code. If the data mapping code maps the entire entry, this indicator is set on. This information is used by update and insert functions to determine if the default entry is to be used for the initial values.
- The lengths of the logical and physical image of the entry.

- The unit of transfer length in bytes.
- If the cursor is active, the following indicators are used:
 - Authority that was saved from the last retrieve authority operation for that data space.
 - Data space used in this activation. A subset of the data spaces the cursor is over can be specified when the cursor is activated.
 - Change bit indicates the data space was changed. Used by the de-activate cursor function to write the changed data to auxiliary storage.

A row in the JMAP contains information about join cursors. There is one JMAP entry per data space under a join cursor. The JMAP table in the cursor header contains the following information:

- A pointer to the data space.
- A pointer to the data space index over this entry.
- The DKEY entry number in the index associated with this entry.
- The self-describing JMAP entry number.
- The JMAP entry number associated with the previous data space (the data space that joined to the current data space).
- Key lengths for the physical join key.
- The number of join key fields.
- A reference count for this position.
- Addressability to this position index control block.
- Addressability to a key buffer, which stores the current key.
- Addressability to a work key buffer used for searching the index.
- An index check count used to determine if the index over this position has changed since positioning.
- Current positioning information (ORD number, ENTRY address).
- Addressability to the data space entry image stored in the cursor header.

- Run time statistics, such as prime and spin.
- Activation statistics such as index placed in use.
- Addressability to join key mapping code that map the fields from the cursor intermediate buffer (CRIB) into a buffer where the key fields are contiguous.

When the cursor has selection specified with it, there is an additional area allocated in the cursor header to describe the associated selection. DB#CRCL is allocated when there is either a cursor for each data space selection or group-by selection is specified. DB#CRCL contains the following information:

- Addressability to the DB#SEL table associated with the cursor per/DS selection.
- Addressability to the DB#SEL table associated with group-by selection.
- Address of merged fast-search array.
- Number of entries in the merged fast-search array.
- Maximum size of Boolean result stacks associated with all the cursor for each data space cursor intermediate key selection routines (selection by key fields only).
- Maximum size of Boolean result stacks associated with all the cursor for each data space noncursor intermediate key selection routines (selection on fields other than key fields and possible key fields).

When selection is specified, DB#SEL is associated with each selection template specified. There is one SEL table allocated for each data space under the cursor when selection is specified for at least one data space. Since selection is not mandatory for each data space, the unused allocated space contains all zeros.

When group-by selection is specified on a cursor, an area in the cursor header is allocated to store DB#GRP. DB#GRP contains information about the group-by selection function which must be performed, addressability to default group-by primitives, addressability to default logical key fields, and addressability to the sets of mapping code possible with group-by selection. The three sets of possible mapping codes include:

- Group-by primitive processing.
- Group-by derived field operations.
- Group-by output mapping code.

A cursor can be either a permanent or temporary object. Before a cursor can be used, it must be activated. Once activated, the data spaces and index (if an index is used) are marked in use to assure their continued existence and for recovery (see *In-Use Table* on this page for additional information). An activate cursor operation attaches the cursor to the activating process by storing the process identification and the current IPL number into the cursor. This action restricts the use of the cursor to the activating process. The only instructions that another process can issue to operate on an active cursor are the Create Duplicate Object and Materialize Cursor Attributes instructions.

De-activation of a cursor detaches the cursor from the process, transfers locks to the associated commit block, and restores the cursor to its original condition. Whenever process termination occurs, any active cursors are de-activated by data base process termination clean-up routines. If an abnormal machine termination occurs, an implicit de-activation is performed the next time the cursor is activated.

In-Use Table

The in-use table is used for recovery purposes by keeping track of the data spaces being used and the data space indexes while they are being built. This table is contained in a permanent preallocated segment. The format of the in-use table is shown in Figure 2-11.

SID Group Header			
Status Bits			
Data Base Level Number			
In-Use Table Forcing Information			
Object @	Object Type	Object Status	Use Count
Object @	Object Type	Object Status	Use Count
			0 (free)
Object @	Object Type	Object Status	Use Count

Figure 2-11. In-Use Table Format

STRUCTURE

The following is a list of the modules in data base management and the function that each module performs. This list also shows how the module is invoked.

#DBACR Activate Cursor

Function: Ensures that a cursor is available to a process and activates the cursor.

How Invoked: Activate Cursor instruction.

#DBAPINC Data Base Maintenance Routine to Apply Increment to Data Space

Function: Applies an increment to the maximum number of entries value for a data space. Utilizes the data base common function **#DBINCRM** to perform the actual increment.

How Invoked: Within this component.

#DBASMEH CSEH for 80, 81, and 82 Microprogramming Exceptions

Function: Traps 80, 81, and 82 exceptions and returns to the next microprogramming instruction.

How Invoked: Through exception management.

#DBASYER Signal Asynchronously Detected Exception During Build Index

Function: Signals the appropriate exception if concurrent activity against the index causes the tree to reach an invalid state or if the concurrent logging SID overflows during a build data space index operation.

How Invoked: Within this component.

#DBBDVAL Build Default Value Array

Function: Loops through the JMAP entries of a join cursor, constructing a 32-bit array indicating which positions contribute default values.

How Invoked: Within this component.

#DBBLDCR Allocate Storage and Build Cursor Header

Function: Allocate storage for and initialize the appropriate fields in the cursor header.

How Invoked: Within this component.

#DBBLDER Signal Select/Omit or Duplicate Key Exception During Build Index

Function: Signals a select/omit or duplicate key exception early if more than 20 errors of one type are detected prior to the catch-up phase of build index.

How Invoked: Within this component.

#DBBRING Bring Data Space Entries

Function: Initiates paging operations as required to page in currently needed data space entries and anticipate future needs.

How Invoked: Data base common function invoked from modules that locate and/or retrieve data space entries.

#DBBUILD Perform Initial Data Space Index Build

Function: Builds a data space index tree by referencing all data space entries under the index and inserting the appropriate keys into the tree.

How Invoked: Within this component.

#DBCALCS Calculate Data Space Size

Function: Calculates the appropriate new size of a data space based on its current size and its attributes.

How Invoked: Within this component.

#DBCATCH Perform Data Space Catch-Up

Function: Applies logged key changes to a data space index at the end of a concurrent build or while activating a delayed maintenance data space index.

How Invoked: Within this component.

#DBCATCY Perform a Build Index Catch-up Cycle

Function: Apply a specified portion of the delayed maintenance logging SID to the DS index or Cursor Key index.

How Invoked: Within this component.

#DBCATDK Handle Duplicate Key

Function: Process duplicate keys encountered during a catch-up cycle.

How Invoked: Within this component.

#DBCATSO Handle Select/Omit Error

Function: Processes select/omit errors encountered during a catch-up cycle.

How Invoked: Within this component.

#DBCCR Create Cursor

Function: Creates a cursor and links it to a data space or data space index.

How Invoked: Create Cursor instruction.

#DBCCREH Create Cursor CSEH

Function: Performs the required recovery when an exception is signaled during a create cursor operation.

How Invoked: Through exception management.

#DBCDS Create Data Space

Function: Creates a data space according to the description provided.

How Invoked: Create Data Space instruction.

#BCDSEH Create Data Space CSEH

Function: This CSEH is enabled by create data space when storage is allocated. This CSEH deallocates the storage that create data space allocated.

How Invoked: Through exception management.

#DBCHNEH Data Space Index Chain CSEH

Function: Detects microprogramming addressing exceptions caused by a reference to nonexistent data space index.

How Invoked: Through exception management.

#DBCINDS Add a DS Index Block to Each Data Space

Function: Build DS index blocks for an index being created and insert them into the header of each data space being updated.

How Invoked: Within this component.

#DBCINEH Create Data Space Index CSEH

Function: Performs the required recovery for an exception that occurred during a create data space index operation.

How Invoked: Through exception management.

#DBCINX Create Data Space Index

Function: Creates a new data space index over one or more existing data spaces.

How Invoked: Create Data Space Index instruction.

#DBCKDUP Check for Duplicate Key

Function: Distinguishes between pseudoduplicate keys and genuine duplicate keys and optionally signals the appropriate exception.

How Invoked: Within this component.

#DBCKICU Build Reserved Key Index

Function: Reserves keys during the build index function if there is a possibility of a decommit reinserting those keys.

How Invoked: Within this component.

#DBCKIJO Read Journal to Build the Reserved Key Index

Function: Reads the journal to simulate a decommit during the build index function and to reserve the keys.

How Invoked: Within this component.

#DBCKSGT Check and Recover a Data Space SID Group Table

Function: Performs recovery operations on the SID group table and related fields of a data space including clean-up operations for partial reset or partial extend.

How Invoked: Within this component.

#DBCLNLD Perform Load/Dump Cleanup

Function: Performs load/dump cleanup of certain data space header fields.

How Invoked: Within this component.

#DBCLONE Clone Segment

Function: Makes a copy of a portion of a segment.

How Invoked: Within this component.

#DBCLRSG Clear Segment

Function: Clears a portion of a segment.

How Invoked: Within this component.

#DBCNVDS Convert Data Space

Function: Performs necessary conversion of data spaces during IPL/recovery and load/dump.

How Invoked: Within this component.

#DBCNVEH Conversion Error Feedback CSEH

Function: Completes conversion error feedback information in the insert sequential option template.

How Invoked: Through exception management.

#DBCPYRC Complete Copy Data Space Entry

Function: Completes the functions required when using a Copy Data Space Entry instruction to copy a data space to itself.

How Invoked: Within this component.

#DBCRIPL Validate Cursor Internals During Segment Identifier Wrap

Function: Validates references to separate segment groups from a cursor, verifies the existence of all objects referenced by the cursor, verifies the existence of all segment groups that comprise the cursor, and detects damage within the cursor.

How Invoked: Segment identifier wrap interface.

#DBDACR De-activate Cursor

Function: Interfaces with the de-activate cursor common function.

How Invoked: De-activate Cursor instruction.

#DBDCR Destroy Cursor

Function: Destroys the specified cursor, removes all context and user profile references to the cursor, and frees all resources associated with this cursor.

How Invoked: Destroy Cursor instruction.

#DBDCTEH Deleted Entry Count CSEH

Function: Sets deleted entry count flags of the data space as suspicious and unreliable if an error occurs while deleting/restoring an entry.

How Invoked: Through exception management.

#DBDDS Destroy Data Space

Function: Destroys the specified data space, removes all user profile and context references to the data space, and frees all resources associated with this data space.

How Invoked: Destroy Data Space instruction.

#DBDEFER Defer Correction of Deleted Entry Count

Function: Defers or corrects the deleted entry count in the data space header.

How Invoked: Within this component.

#DBDEFLT Initialize Fields with Default Values

Function: Initializes fields with the appropriate default values according to field type.

How Invoked: Within this component.

#DBDELCT Validate/Recover Deleted Entry Count for Data Space

Function: Determines number of deleted entries in data space and assures that the deleted entry counts in the data space header and each data SID header are correct.

How Invoked: Within this component.

#DBDELEN Delete Data Space Entry

Function: Deletes an entry from the data space.

How Invoked: Within this component.

#DBDELIM Data Base Common Function to Delete Data Space Entries

Function: Deletes data space entries.

How Invoked: Within the data base and journal components.

#DBDEREH Derived Mapping Exception Handler

Function: Handle exceptions that occur while executing derived field mapping.

How Invoked: Within this component.

#DBDERKY Derive Logical Key

Function: Maps derived keys.

How Invoked: Within this component.

#DBDINX Destroy Data Space Index

Function: Destroys the specified data space index, removes all user profile and context references associated with the data space index, and frees all resources associated with the index and also removes any references from any associated space.

How Invoked: Destroy Data Space Index instruction.

#DBDISHR Dispose of Hold Record

Function: Disposes of the data space entry hold record lock and releases it to the system or to a commit block.

How Invoked: Within this component.

#DBDIXEH Destroy Mini-Indexes

Function: Destroys mini-indexes created during a build index in the event of an exception.

How Invoked: Within this component.

#DBDKFEH Duplicate Key Feedback CSEH

Function: Completes duplicate key feedback information in the option template of the Insert Sequential Data Space Entries instruction.

How Invoked: Through exception management.

#DBDKYEH Delete Key CSEH

Function: Restores all modified data space indexes to their prior state by inserting all recently deleted keys.

How Invoked: Within this component.

#DBDMGCR Damage a Cursor

Function: Set hard damage to a cursor.

How Invoked: Within this component.

#DBDMLX Extend Delayed Maintenance Logging SID

Function: Extends the logging SID when more space is required to record a data space index modification.

How Invoked: Within this component.

#DBDMLOG Logs a Data Space Index Update

Function: Records a pending data space index binary tree modification so that the data space index can be brought up to data at a later time.

How Invoked: Within this component.

#DBDPKEH Duplicate Key CSEH

Function: Suppresses tentative ordinal numbers within exception data.

How Invoked: Through exception management.

#DBDQDSE Remove Entry Lock From Queue

Function: Removes a data space entry lock from the cursor's locked entry queue.

How Invoked: Within this component.

#DBDSIPL Validate/Recover Data Space

Function: Detects internal damage within a data space during install and load and after a system failure, restores the data space to a useable condition, and verifies the existence of all segment groups that constitute the data space.

How Invoked: Within this component.

#DBDSI12 Converts Data Space Index to Release 2

Function: Converts the data space index from release 1 to release 2 format.

How Invoked: Within this component.

#DBDSR12 Converts Data Space to Release 2

Function: Converts the data space from release 1 to release 2 format.

How Invoked: Within this component.

#DBDSSEL Verify Selection Templates and Generate Selection Code

Function: Generate data space selection code.

How Invoked: Within this component.

#DBENDSE Ensure Data Space Entries

Function: Places on auxiliary storage all data space entries that have been modified through this cursor since the last ensure operation or since cursor activation.

How Invoked: Ensure Data Space Entries instruction.

#DBENTAD Calculate Virtual Address of a Data Space Entry

Function: Calculates the virtual address of a data space entry given the data space virtual address and the ordinal number of the data space entry.

How Invoked: Within the data base and journal components.

#DBFORDS Force a Data Space and any Eligible Indexes to Auxiliary Storage

Function: Saves the data space on auxiliary storage, clears change flags within all affected data space indexes, and writes to auxiliary storage only those indexes in which all change flags have been cleared for all data spaces in those indexes.

How Invoked: Within this component and by the ensure object function.

#DBFORIN Force Inserted Entries

Function: Forces all inserts for a single data space. This function is invoked if ensure is active or a user has the write operation specified on an insert operation.

How Invoked: Within this component.

#DBFORSG Force all Data Spaces Under the Cursor

Function: Places on auxiliary storage all data space entries that have been modified by a cursor operation since the last ensure operation or the activation of the cursor.

How Invoked: Within this component.

#DBFORUP Force Data Space Updates

Function: Forces all updated data space entries to nonvolatile storage (any data segment group containing updated data space entries).

How Invoked: Within this component.

#DBFRCEN Force Locked Entry

Function: Forces a single data space entry to nonvolatile storage.

How Invoked: Within this component.

#DBFRCIX Force Data Space Index

Function: Causes a data space index to be forced to disk by forcing the changed data spaces referenced by that data space index.

How Invoked: Within this component.

#DBFXBDY Fix Group Boundary

Function: Copies a data space entry spanning a group boundary into a replacement buffer.

How Invoked: Within this component.

#DBHDCEH Damage Cursor CSEH

Function: Damages the cursor if an exception occurs during the time this CSEH is enabled.

How Invoked: Within this component.

#DBIKYEH Insert Key CSEH

Function: Restores all modified data space indexes to their prior state by removing all recently inserted keys.

How Invoked: Within this component.

#DBINBDY Insert Data Space Entry Spanning SID Group Boundary

Function: Inserts a single data space entry that crosses an SID group boundary.

How Invoked: Within this component.

#DBINCRM Data Base Common Function to Apply Increment to Data Space

Function: Applies increment to data space function.

How Invoked: Within the data base and journal components.

#DBINFER Induce an Invalid Entry Status Byte for Recovery Purposes

Function: Deliberately invalidates status byte for recovery purposes.

How Invoked: Within this component.

#DBINIT1 Data Base IPL and Recovery Phase 1

Function: Performs all possible cleanup functions for data base objects in use at the time of an abnormal system termination. This is phase 1 of the data base recovery.

How Invoked: Within this component.

#DBINIT2 Data Base IPL and Recovery Phase 2

Function: Performs lingering cleanup functions for data base objects affected by journal and commit recovery. This is phase 2 of the data base recovery.

How Invoked: Within this component.

#DBINIWA Randomize the Invoker's Automatic Variables

Function: Reach into the invoker's automatic storage and randomize the variables.

How Invoked: Within this component.

#DBINJDS Handle Injured Data Space

Function: Identifies a damaged data space, and discards all linkage to associated indexes.

How Invoked: Within this component.

#DBINSDR Data Base Maintenance Routine to Insert Default or Deleted Entries

Function: Inserts default or deleted entries into a data space. Utilizes the data base common function #DBINSIM to perform the actual inserts.

How Invoked: Within this component.

#DBINSEN Insert Data Space Entry

Function: Inserts a data space entry into the specified data space.

How Invoked: Insert Data Space Entry instruction.

#DBINSEQ Insert Sequential Data Space Entries

Function: Inserts data space entries into the specified data space.

How Invoked: Insert Sequential Data Space Entries instruction.

#DBINSIN Insert Data Space Entries

Function: Performs the data base common function to insert data space entries into the data space.

How Invoked: Within the data base and journal components.

#DBIPLEH Tolerate Selected Page Reference Exceptions that Occur During IPL

Function: Provides tolerance and resume point support for page reference exceptions because of object damage.

How Invoked: Through exception management.

#DBISRCH Search Data Space Index

Function: Examines binary tree contents to ensure that the key utilized to locate the data space entry during the Set Cursor instruction still identifies the same data space entry.

How Invoked: Within this component.

#DBIVLEH CSEH to Invalidate Data Space Indexes

Function: Responds to unexpected exceptions encountered by invalidating all modified data space indexes affected by the data space being populated.

How Invoked: Through exception management.

#DBIVXEH Invalidate a Data Space Index

Function: Invalidates a specific data space index in the event an exception occurs while manipulating the data space index.

How Invoked: Within this component.

#DBIXCEH CSEH to Invalidate a Data Space Index

Function: Responds to errors signaled during a machine index operation by invalidating the data space index involved in the operation.

How Invoked: Through exception management.

#DBIXCHN Perform Modification of Data Space Indexes

Function: Performs insert key(s), delete key(s), empty index requests, and invalidation requests when performing modifications to the underlying data spaces.

How Invoked: Within this component.

#DBIXFEH Data Space Index Full CSEH

Function: Invalidates full indexes that utilize release 1 format (2-byte) node structure.

How Invoked: Through exception management.

#DBIXGES Estimate Size of Data Space Index

Function: Estimates the number of index entries between the low and high points of a key range.

How Invoked: Within this component.

#DBIXIPL Validate/Recover Data Space Index

Function: Detects internal damage within a data space index during install and load or following a system failure, invalidates the index if required, and verifies the existence of all segment groups that constitute the data space index.

How Invoked: Within this component.

#DBIXUEH CSEH to Remove DS Index from In-Use Table

Function: Removes DS index from the in-use table.

How Invoked: Within this component.

#DBIXUSE Set Data Space Index Concurrent Log Bit

Function: Set the concurrent log bit in the data space index header if the data space index is being taken out of use and there is no other user of this data space index.

How Invoked: Within this component.

#DBIXVDS Verify Data Space and Data Space Index Properly Address each Other

Function: Examines the DKEY table of a data space index and the data space index directory blocks of a data space to verify that they properly address each other.

How Invoked: Within this component.

#DBJTLEH CSEH to Tolerate Journal Errors

Function: Tolerates the return feedback information on journal errors.

How Invoked: Within the data base and load/dump components.

#DBLABRG Look-Ahead

Function: Performs look-ahead of data space entries.

How Invoked: Within this component.

#DBLCTKY Locate Key Candidate in Data Space Index

Function: Locates a key candidate using a search key and rule option.

How Invoked: Within this component.

#DBLGDNT Log Information About a Faulty Entry Status Byte into the VMC Log

Function: Logs information associated with a faulty entry status byte into the VMC log.

How Invoked: Within this component.

#DBLKMAP Record the First Execution of a Module, Entry Point, or Function

Function: Records the address, name, and compile date of the calling module and then no-ops the call instruction. Used to identify newly linked modules and to identify the first execution of rarely executed paths.

How Invoked: Within this component.

#DBLOGGER Log Possible Error Keys During Data Space Index Build

Function: Invokes the delayed maintenance logging function to log the key. If the error threshold is reached, then sets the error return flag.

How Invoked: Within this component.

#DBMAINT Data Base Maintenance

Function: Performs special maintenance operations on data base objects. The options supported are:

- Rebuild Index: Rebuild the index tree of an invalid index based upon the data spaces pointed to by the index.
- Invalidate Index: Mark a data space index as invalid making the index un-usable until it is rebuilt, and reclaim unused space.
- Reset Data Space: Delete all entries in a data space, reclaim unused space, and free all ordinal numbers previously assigned.
- Apply Increment to Data Space: Apply the increment to the maximum number of entries value for a data space.
- Insert deleted entries.
- Insert default entries.

How Invoked: Data Base Maintenance instruction.

#DBMAIVI Validate Data Space Index Chains

Function: Ensures that the data space index chain of a data space is usable and optionally ensures that the data space indexes over the data space are in a state that permits the currently executing instruction to complete.

How Invoked: From modules that must use the data space index chain and do not perform their own checking.

#DBMAPEH Conversion/Mapping CSEH

Function: Performs the required recovery because of decimal data and overflow exceptions caused by data fields operated on by data base generated mapping code.

How Invoked: Through exception management.

#DBMATCR Materialize Cursor Attributes

Function: Determines if the user requests materialization of the statistics or the creation template for the specified cursor, and invokes the appropriate materialization function.

How Invoked: Materialize Cursor Attributes instruction.

#DBMATDS Materialize Data Space Attributes

Function: Determines if the user requests materialization of the statistics or creation template and moves the requested information into a space object.

How Invoked: Materialize Data Space Attributes instruction.

#DBMATIX Materialize Data Space Index Attributes

Function: Determines if the user requests materialization of the statistics or creation template and invokes the appropriate materialization function.

How Invoked: Materialize Data Space Index Attributes instruction.

#DBMCLEH CSEH to Discard Partially Inserted Data Space Entries

Function: Discards all partially inserted data space entries from the specified data space.

How Invoked: Through exception management.

#DBMDSAT Modify Data Space

Function: Modifies the data space attributes.

How Invoked: Within this component.

#DBMERGE Merge Mini-Indexes into a Data Space Index

Function: Merges mini-indexes into the data space index during build index.

How Invoked: Within this component.

#DBMIVAL Invalidate a Data Space Index

Function: Invalidates a data space index.

How Invoked: Within this component.

#DBMODIM Modify Data Space Entry

Function: Updates data space entries.

How Invoked: Within the data base and journal components.

#DBMODIX Modify Attributes of a Data Space Index

Function: Changes an attribute of a data space index that can be modified.

How Invoked: Modify Data Space Index Attribute instruction.

#DBMONRL Release a Seize and Wait

Function: Releases a seized object. Performs a wait operation if some other process is attempting to access the object.

How Invoked: Within this component.

#DBMONSZ Seize an Object and Save the Hold Record Address

Function: Seizes an object and saves the address of the hold record so that the hold record can be tested periodically to determine if another process is attempting to access the object.

How Invoked: Within this component.

#DBMPSEL Derived Field Mapping

Function: Performs derived field mapping and selection for each DMAP entry.

How Invoked: Within this component.

#DBMRBLD Rebuild a Data Space Index

Function: Rebuilds a data space index for data base maintenance.

How Invoked: Within this component.

#DBMRSET Data Base Maintenance Routine to Reset a Data Space

Function: Performs the data base maintenance option to reset the data space.

How Invoked: Within this component.

#DBMR6F6 Merge Fast Search Arrays into a Single Array

Function: Process ranges from one or more fast search arrays to produce a single merged array with ordered disjointed ranges.

How Invoked: Within this component.

#DBMUSEH Remove Objects from In-Use

Function: CSEH to remove the specified objects from the in-use table in the event of an exception.

How Invoked: Within this component.

#DBORDNB Determine Ordinal Number

Function: Determines the ordinal number associated with the given virtual address of a data space entry.

How Invoked: Within this component.

#DBPOSCR Restore a Cursor Position

Function: Restores the position of a cursor.

How Invoked: Within this component.

#DBRELEH Release Seized Objects CSEH

Function: Releases the objects listed in the designated seize/release parameter block.

How Invoked: Through exception management.

#DBRESEQ Sequentially Retrieve Data Space Entries

Function: Sequentially retrieves multiple data space entries and sets the cursor to address the final entry retrieved.

How Invoked: Retrieve Sequential Data Space Entries instructions.

#DBRESET Logically Reset a Data Space

Function: Identifies a data space as totally or partially reset and destroys any unused data SIDs.

How Invoked: Within this component.

#DBRETEN Retrieve Data Space Entry

Function: Maps the data space entry designated by the cursor into the user's interface buffer.

How Invoked: Retrieve Data Space Entry instruction.

#DBRLSEN Release Data Space Entries

Function: Releases data space entries (unlocks either the first entry or all entries currently locked to a specified active cursor). The number of entries unlocked is determined by the release data space entries option.

How Invoked: Release Data Space Entries instruction.

#DBRMDSK Removes Data Space Keys from a Data Space Index

Function: Removes all keys for a data space from a data space index.

How Invoked: Within this component.

#DBRSIPL Recover from Reset

Function: Performs IPL-time recovery for a partial reset operation against SID number 1 of a data space.

How Invoked: Within this component.

#DBRSQMN Retrieve Sequential Mainline

Function: Retrieves multiple data space entries or group-by results.

How Invoked: Within this component.

#DBRSTDS Data Base Common Function to Perform a Reset of a Data Space

Function: Fully resets a data space.

How Invoked: Within the data base and journal components.

#DBSCAEH Store and Set Computational Attributes Exception Handler

Function: The computational attributes are reset from the exception data.

How Invoked: Within this component.

#DBSELEH Generated Selection Code Exception Handler

Function: Sets the exception type and returns to the next sequential instruction of the invoker of the generated selection code if a data related exception occurs during the execution of the generated selection code.

How Invoked: Within this component.

#DBSELIX Select Affected Data Space Indexes

Function: Selects the data space indexes that are affected by the modification of an underlying data space.

How Invoked: Within this component.

#DBSETCR Set Cursor

Function: Causes the cursor to address an entry in a data space as specified by the option list and the search key. The address of the entry is stored in the cursor. If requested, the logical key is returned in the area specified.

How Invoked: Set Cursor instruction.

#DBSOEH Select/Omit CSEH

Function: Tolerates exceptions associated with a select/omit data space index that occurred during the execution of a user exit program.

How Invoked: Through exception management.

#DBSPCHK Run-Time Deleted Segment Group Spanning Data Space Entry Checks

Function: Checks the segment group spanning data space entries to assure that consistency is maintained between the data space and data space index.

How Invoked: Within this component.

#DBSPNBG Detect and Process the Data Spaces That Were Exposed to the Deleted Span Segment Indicator

Function: Attempts to correct any damage caused by the indicator that inadvertently deletes data space entries that cross segment group boundaries.

How Invoked: Within this component.

#DBSRCHN Search DSPI Chain During Recovery and Load/Dump

Function: Performs validity checks on the DSPI chain of a data space during load/dump and IPL recovery.

How Invoked: Within this component.

#DBSTUFF Build Keys from Data Spaces and Insert Them into an Index

Function: Performs appropriate select/omit and build key processing on data space entries and inserts the entries into a data space index or mini-index during a build index operation.

How Invoked: Within this component.

#DBSTUFX Put Keys into Index from Parent Index

Function: Generates keys from an existing index and inserts them into the new index.

How Invoked: Within this component.

#DBSTUSB Build Key of Entry Crossing SID Group Boundary

Function: Performs appropriate select/omit and build key processing for a data space entry that crosses an SID group boundary.

How Invoked: Within this component.

#DBTERM Data Base Process Termination Routine

Function: De-activates any cursors that are active at process termination.

How Invoked: Process termination.

#DBTMPPIX Manage Retain Status of a Data Space

Function: Modifies the retain status of the data space to account for the presence of a temporary index referenced from the DS index chain.

How Invoked: Within this component.

#DBUCOPY Copy Data Space Entries

Function: Copies data space entries from one data space to another (or the same) data space according to the specified options.

How Invoked: Copy Data Space Entries instruction.

#DBUCSEH Copy Data Space Entries CSEH

Function: Handles errors that occur during a Copy Data Space Entries instruction. When source and receiver data spaces are the same, backs up the data space to its prior state.

How Invoked: Through exception management.

#DBUGDEL Report an Inconsistent Deleted Entry Count

Function: Flags and reveals an inconsistent deleted entry count of a data space.

How Invoked: Within this component.

#DBUNBLD Unbuild Composite Key

Function: Extracts individual fields from a composite key and reverses any key building operations to obtain the previous key value.

How Invoked: Within this component.

#DBUNDEL Update a Deleted Entry

Function: Updates a deleted data space entry.

How Invoked: Within this component.

#DBUPDEN Update Data Space Entry

Function: Updates the locked data space entry addressed by the head of the locked entry queue associated with the cursor.

How Invoked: Update Data Space Entry instruction.

#DBUPDIM Update a Nondeleted Entry

Function: Updates a nondeleted data space entry.

How Invoked: Within this component.

#DBVERIP Verify Internal Pointer

Function: Verifies that an internal segment pointer contains an undamaged segment identifier.

How Invoked: Within this component.

#DBVERPT Verify External Pointer

Function: Verifies that an external segment pointer references an existing segment group.

How Invoked: Within this component.

#DBVERSD Data Base Object Router for Segment Identifier Wrap

Function: Determines the object type during segment identifier wrap processing and invokes the appropriate object-specific validation routine.

How Invoked: Through the segment identifier wrap interface.

#DBVYJN Verify Cursor Join Position

Function: Verifies the current position of a join cursor to ensure it is still a valid composite join position.

How Invoked: Within this component.

#DBVYKY Verify Key Presence in Data Space

Function: Determines if the key corresponding to the data space entry is present in the input data space index.

How Invoked: Within this component.

#DBVIPTR Verify a SID pointer

Function: Verify the input pointer for accuracy. If an erroneous pointer is detected, replace the pointer with the standard value representing a lost SID.

How Invoked: Within this component.

#DBVKCNT Verify MI-Specified Key Count of Byte Length or Number of Key Fields

Function: Given a logical key count, consisting of either a byte length or key field count, verifies the contents and returns the length of the logical key including fork characters.

How Invoked: Within this component.

#DBVRSEL Verify Selection Template

Function: Verify that the selection template is internally consistent.

How Invoked: Within this component.

#DBXBIES Calculate Value For Extending Data Space Indexes

Function: Given the current data space index size and an optional minimum new size, calculates a new size that fits desired allocation rules.

How Invoked: From modules that create or extend a data space index.

#DBXBINX Build Data Space Index

Function: Builds or brings up to date the binary tree (internal machine index) portion of a data space index.

How Invoked: Within this component.

#DBXBLKY Build Composite Key

Function: Builds a composite key according to the input parameters.

How Invoked: Within this component.

#DBXCBEH Critical Data Base Error CSEH

Function: Causes CSEH to write a data base object image to the VMC log in case of an unrecoverable error in manipulating the object, and to stop machine processing so that data base IPL recovery can attempt to recover the error.

How Invoked: Through exception management.

#DBXCIRC Log Circular Data Space Index Directory Block Chains

Function: VMC logs a data space when circular data space index directory block chain is detected and soft damage to the data space occurs.

How Invoked: Within this component.

#DBXCLDS Clear Trailing Area of Data Segment

Function: Discards the entries from the end of a data space.

How Invoked: Within this component.

#DBXCPCR Copy Cursor

Function: Restores object specific fields in the cursor after a create duplicate object operation.

How Invoked: Other VMC component as a result of a create duplicate object operation.

#DBXDACR De-activate Cursor Common Function

Function: De-activates the specified cursor.

How Invoked: Within this component and process termination.

#DBXDDS Destroy Data Space Common Function

Function: Performs common portions of destroy data space for error and IPL recovery.

How Invoked: Within this component.

#DBXDINX Destroy Data Space Index Common Function

Function: Performs common portions of destroy data space index.

How Invoked: Within this component and destroy and create data space index CSEHs.

#DBXDSID Carefully Destroy Data Segment

Function: Verifies the identity of a data space data segment and destroys it.

How Invoked: Within this component.

#DBXFDSH Load/Dump Update Data Space Header

Function: Updates addresses in a loaded data space header and merges the data space index blocks of the overlaid data space if one exists.

How Invoked: From load/dump.

#DBXFIXH Load/Dump Update Data Space Index Header

Function: Updates addresses in a loaded data space index header and establishes pointers between the index and the referenced data spaces.

How Invoked: From load/dump.

#DBXFMAP Data Movement/Conversion Code Generation

Function: Generates field mapping and data conversion code.

How Invoked: Within this component.

#DBXGBDY Parse Entry Spanning a Group Boundary

Function: Locates both portions of a data space entry spanning a SID group boundary.

How Invoked: Within this component.

#DBXGDER Generated Derived Field Mapping

Function: Processes the mapping template to produce derived field mapping.

How Invoked: Within this component.

#DBXIVAL Invalidate Data Space Index

Function: Invalidates an index and signals an event.

How Invoked: Within this component.

#DBXMATK Construct Logical Key

Function: Generates the logical key corresponding to the data space entry.

How Invoked: Within this component.

#DBXMUSE Maintain the Data Base In-Use Table

Function: Records the data base objects that are being used at any one time. This routine performs all IPL and run-time functions that involve the in-use table. These functions include:

- Initialize: Initializes the in-use table (IPL)
- Set: Sets pointer to retrieve first in-use table entry (IPL)
- Retrieve: Retrieves next in-use table entry (IPL)
- Find: Determines if an object is in-use (run time)
- Increment: Places an object in-use or increases the count (run time)
- Decrement: Removes an object from in-use or decreases the count (run time)
- Ensure: Ensures that the in-use status of all objects is saved on secondary storage (run time)
- Signal Exception: Signals an object not eligible exception (run time)
- Retain: Flags an object as a candidate for recovery processing

How Invoked: Within this component and from load/dump.

#DB5NDEF Verify Join Definition Template

Function: Verifies the join input template, then builds the join portion of a cursor.

How Invoked: Within this component.

#DB5NPOS Set Secondary Position of a Join Cursor

Function: Sets the secondary position of a join cursor.

How Invoked: Within this component.

#DB6BDEF Verify Group-by Definition Template List and Build Related Group-by Structures

Function: Builds the intermediate buffer description for group-by results and generates all primitive processing code.

How Invoked: Within this component.

#DB6BMAP Verify Group-by Mapping

Function: Verifies group-by mapping code and generate group-by intermediate mapping code.

How Invoked: Within this component.

#DB6BSEL Verify Group-by Selection Template and Generate Group-by Selection Code

Function: Verifies and generates Selection code for the group-by intermediate buffer.

How Invoked: Within this component.

#DB6MSEL Group-by Derived Field Mapping and Selection

Function: Performs group-by derived field mapping, selection, and output mapping.

How Invoked: Within this component.

#DB6NSEL Generate Selection Code

Function: Processes the selection template to produce the generated selection code.

How Invoked: Within this component.

#DBXPUSH Allocate Additional Data Space SID Group Table Entries

Function: Allocates more space for SID group table entries in a data space header by relocating any data space index directory blocks in the data space header.

How Invoked: Within this component.

#DBXRINX Load/Dump Network Cleanup

Function: Searches the data space index chain of previously loaded and updated data space header for any blocks with the load/dump flag set to on. Such blocks are deleted and their spaces reclaimed. The data space index chain is also searched for any blocks with the index invalidated flag set to on. An event is signaled for each index pointed to by such a block.

How Invoked: From load/dump.

#DBXRMVI Remove Flagged Data Space Index Blocks from the Data Space Header

Function: Searches the index list of a data space header for any blocks with the remove flag set to on or the load/dump flag set to on, and deletes those blocks.

How Invoked: Within this component.

#DBXRMVT Re-copy Data Space Header to the Data Space Header Segment Group

Function: Copies the duplicated (copied) data space header to the original header and destroys the duplicated segment group.

How Invoked: Within this component.

#DBXSELT Invoke Select/Omit Routine

Function: Maps data space entry fields to the selection buffer and invokes the selection routine to determine if the entry is to be inserted into the data space index.

How Invoked: Within this component.

#DBXSIZE Alter the Size of a Segment Group of an Object

Function: Alters the size of a segment group of an object to the specified size.

How Invoked: Within this component.

#DBXTNDS Extend the Space Allocated for a Data Space

Function: Extends a data space such that it is large enough to hold additional entries.

How Invoked: Within this component.

#DBXUNLK Unlock Data Space Entry

Function: Removes and disposes of the specified hold records from the cursor's locked entry queue.

How Invoked: Within this component.

#DBXVDER Verify Derived Field Mapping Template

Function: Verifies that the derived field mapping template is internally consistent.

How Invoked: Within this component.

#DBXVERX Verify that a Segment Group Does Exist

Function: Verifies that the designated segment group does exist. If the segment does not exist, a designated object is marked damaged.

How Invoked: Within this component.

#DBXVMAP Verify Mapping Template

Function: Verifies that the designated field resides within the data space entry, the type attribute is valid, and the length for the attribute specified is valid.

How Invoked: Within this component.



Independent Index Management

INTRODUCTION

An independent index is a system object used for data storage and retrieval. The index is designed to minimize the storage required for data and the time needed to insert, find, or remove pieces of data. Independent indexes have multiple uses; among these are table searches, sorts, merges, cross-references, and symbol tables.

Independent index management controls the building and maintenance of independent indexes. Independent index management uses a machine index for the storage of data and uses machine index management to perform the operations on data contained in the indexes.

Independent index management supports the following System/38 instructions:

- Create Independent Index
- Destroy Independent Index
- Find/Remove Independent Index Entry
- Insert Independent Index Entry
- Materialize Independent Index Attributes
- Modify Independent Index

Figure 3-1 shows an overview of these independent index management functions.

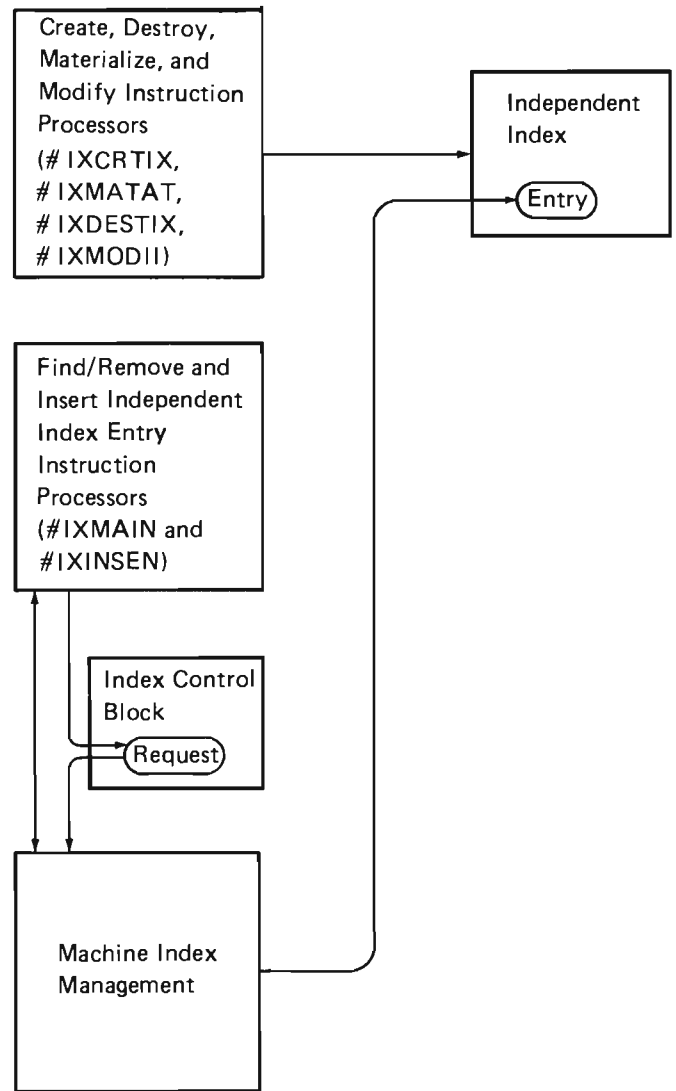


Figure 3-1. Independent Index Management Overview

Create Independent Index

Module #IXCRTIX is invoked by the supervisor link (SVL) router as a result of a Create Independent Index instruction. #IXCRTIX copies the index description template into an internal area and calls #CFCRTO1 to validate the template.

#IXCRTIX then invokes #SMSGCRT to allocate the storage, and #IXCRTIX initializes the encapsulated program architecture (EPA) and object specific headers and the base page of the index. #CFCRTO2 is then called to insert addressability to the index into a context (if requested) and the user profile (if the index is permanent). The entire object is written to auxiliary storage by #SVE8PPR. The system pointer is initialized and returned to the caller.

Destroy Independent Index

Module #IXDESTIX is invoked by the SVL router as a result of a Destroy Independent Index instruction. #IXDESIX invokes #CFOCHKR to seize and validate the index. #IXDESIX then invokes #CFDESTO to destroy the index (headers and machine index) and to remove addressability from any context and user profile containing addressability to the index. Finally, #IXXDEST is invoked to destroy any secondary index segments.

Find/Remove Independent Index Entry

Module #IXMAIN is invoked by the SVL router as a result of both Find Independent Index Entry and Remove Independent Index Entry instructions. #IXMAIN validates the instruction and then invokes #CFOCHKR to seize and validate the index. #IXMAIN copies the rule options list into an area in the invocation work area (IWA), validates the list, and then builds an index control block (IXCB) according to the rule options list as follows:

Generic Request	Rule Option (Hex)	Machine Index Function		
		First EXCB (Find or Remove)	Subsequent EXCBs (Find Only)	Subsequent EXCBs (Remove Only)
Generic Equals	0001	Find low	Find next	Find low else next
Strictly Greater Than	0002	Find generic next	Find next	Find low else next
Strictly Less Than	0003	Find generic prior	Find generic prior	Find generic prior
Greater Than or Equal	0004	Find low else next	Find next	Find low else next
Less Than or Equal	0005	Find high else prior	Find generic prior	Find generic prior
First	0006	Find lowest	Find next	Find low else next
Last	0007	Find highest	Find generic prior	
Between (inclusive)	0008	Find low else next or find high else prior	Find next or find generic prior	Find low else next or find generic prior

#IXMAIN then invokes the machine index function (#IXEXCB) to perform the index operation. When an entry is located, the status flags are checked for error conditions. The following sequence is then repeated until all occurrences of the entry have been located:

1. If a remove entry operation is specified, the entry just found is removed from the index.
2. An internal return count is incremented.
3. Initialize the IXCB for the next find according to the preceding rule options, and execute the control block.
4. Check the status flag for an error condition.

When all entries have been found, the return count is placed in the option list and the index is released.

Insert Independent Index Entry

Module #IXINSEN is invoked by the SVL router as a result of an Insert Independent Index Entry instruction. #IXINSEN invokes #CHOCHKR to seize and validate the index. #IXINSEN copies the rules options list to an area in the IWA, validates the list, and then builds an IXCB according to the rule options list as follows:

Generic Request	Rule Option (Hex)	Machine Index Function
Insert	0001	Insert
Insert with function replacement	002	Insert
Insert conditional that key not in index	0003	Insert conditionally

For each entry to be inserted, move the argument to an internal buffer, perform the index function (#IXEXCB), check the status flags (extend index if necessary), and increment the return count.

When all entries have been inserted, the return count is placed in the options list and the index is released.

Materialize Independent Index Attributes

Module #IXMATAT is invoked by the SVL router as a result of a Materialize Independent Index Attributes instruction. #IXMATAT invokes #CFOCHKR to seize and validate the index. #IXMATAT invokes #CFMAT02 to copy the data to be materialized from the EPA header. #IXMATAT then copies the data from the object specific header to an internal buffer, and then moves the data (up to the length of the template) from the buffer to the user template. The find count in the object specific header is set to 0 and the index is released.

Modify Independent Index

Module #IXMODII is invoked by the SVL router as a result of a Modify Independent Index instruction. #IXMODII invokes #CFOCHKR to seize and validate the index. #IXMODII copies the index to auxiliary storage. #IXMODII modifies the specified attributes. When all specified attributes are modified, the index is released.

DATA AREAS

Independent Index

An independent index resides in one or two segment groups as shown in Figure 3-2. The first segment contains the following:

- The segment group header
- The EPA header
- An object specific header that contains:
 - Index attributes, values, and pointers
 - Buffer
- A machine index

The second segment is optional and contains the following:

- The segment group header
- The associated space

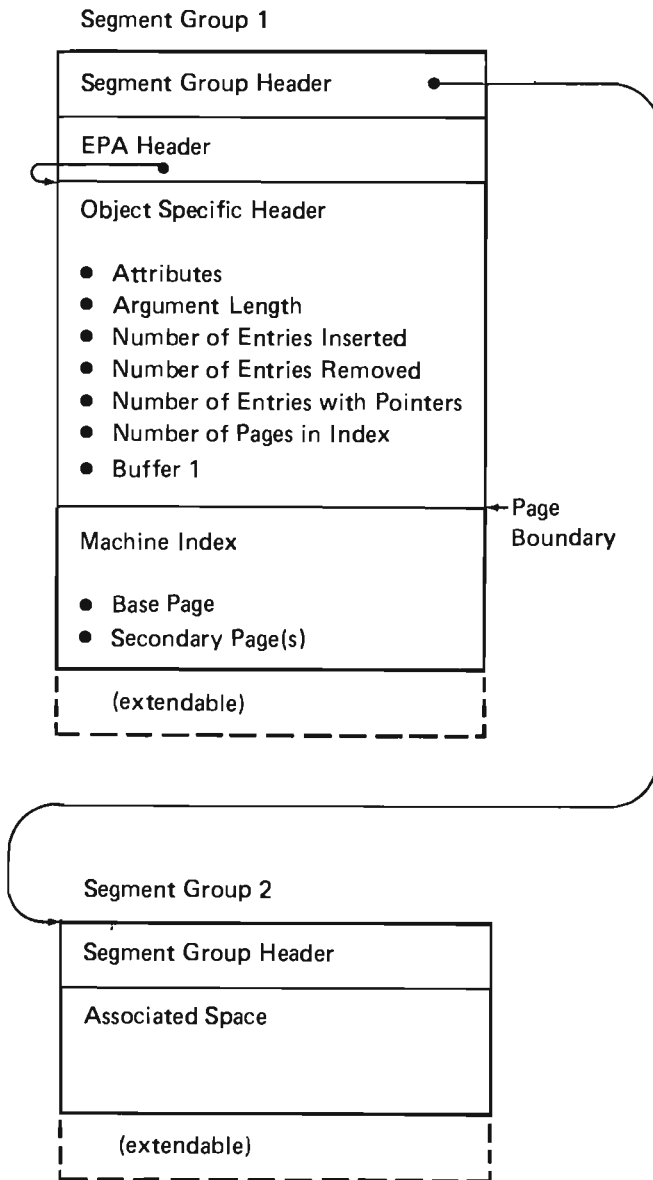


Figure 3-2. Independent Index

Index Description Template

The index description template is used by the create and materialize index functions to build and materialize an index. The template contains:

- Attributes and values
- Creation and recovery options
- Pointers

STRUCTURE

The following is a list of the modules in independent index management and the function that each module performs. The list also shows how the module is invoked.

#IXCRTEH Create Index Component-Specific Exception Handler (CSEH)

Function: Processes exceptions that occur during a create independent index operation.

How Invoked: VMC component.

#IXCRTIX Create Independent Index

Function: Creates an independent index according to the specifications contained in the creation template.

How Invoked: Create Independent Index instruction.

#IXDESIX Destroy Independent Index

Function: Destroys the specified index and removes all references to the index from the system.

How Invoked: Destroy Independent Index instruction.

#IXINSEN Insert Independent Index Entry

Function: Inserts entries into the specified independent index.

How Invoked: Insert Independent Index Entry instruction.

#IXINXEH Index CSEH

Function: Processes exceptions that occur during insert, find, and remove entry operations.

How Invoked: VMC Component.

#IXMAIN Index Main

Function: Mainline index code that processes find and remove index entry options.

How Invoked: Find Independent Index Entry and Remove Independent Index Entry instructions.

#IXMATAT Materialize Independent Index

Function: Materializes the attributes of the specified index.

How Invoked: Materialize Independent Index instruction.

#IXMODII Modify Independent Index

Function: Modifies the specified attributes within the independent index.

How Invoked: Modify Independent Index instruction.

Journal Management

INTRODUCTION

Journal management records the changes made to objects for use in recovery procedures and tracks change activity.

Along with recording the changes made to an object, journal management also records the following information about the object:

- When the change is made.
- What process makes the change.
- The user that makes the change.
- What program makes the change.

Journal management also:

- Records the changes simultaneously on two journal spaces so that if one journal space is damaged, the information can be retrieved from the undamaged journal space.
- Automatically synchronizes the journal port and the journaled object during initial program load (IPL). This also synchronizes the journaled object with all other objects being journaled through the same journal port.
- Allows the user to place entries on the journal. These may be interspersed with entries for object changes.
- Retrieves the entries from the journal by a variety of search criteria.
- Recovers the journaled object from the journal port. This can either involve forward recovery of the object by applying the specified changes, or backward recovery by applying the inverse of the specified changes.

Journal management uses two system objects to implement these functions as shown in Figure 4-1. They are the journal port and journal space.

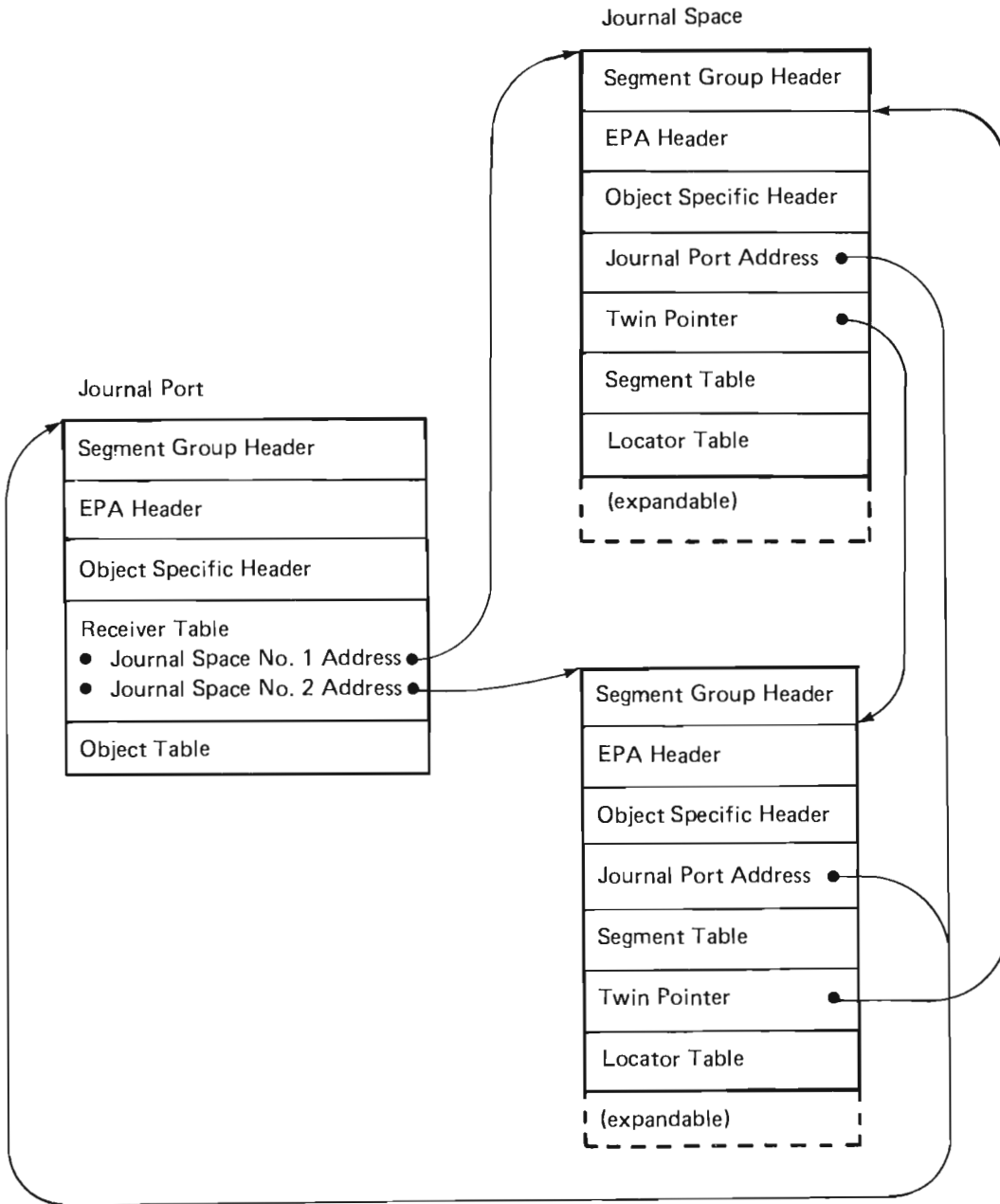


Figure 4-1. Journal Object Relationship

The journal port (Figure 4-2) is the object through which journal entries are routed. It provides a method of linking the journaled objects to the journal spaces. The journal port also provides a definition of the lengths of the various prefix data.

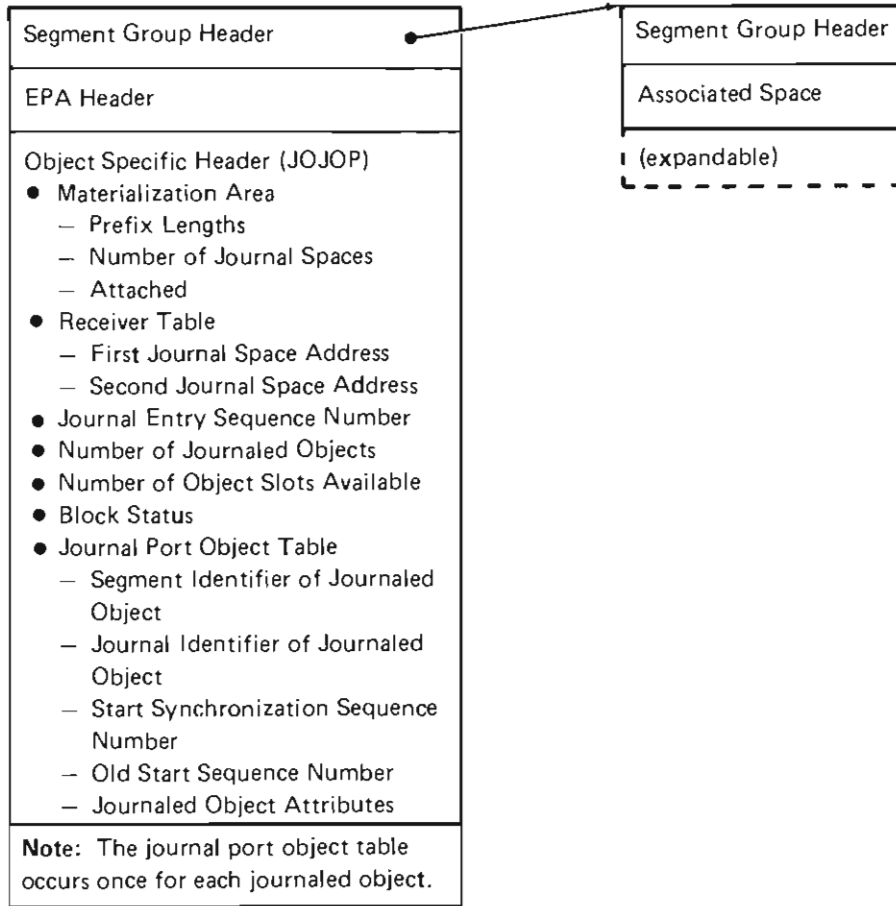


Figure 4-2. Journal Port

The journal space (Figure 4-3) is used to record journal entries. When the journal space is attached to a journal port, entries are placed into the journal space for any change to an object that is being journaled through that journal port. The entries are variable length. Linkage is provided for both forward and backward search, and random finds.

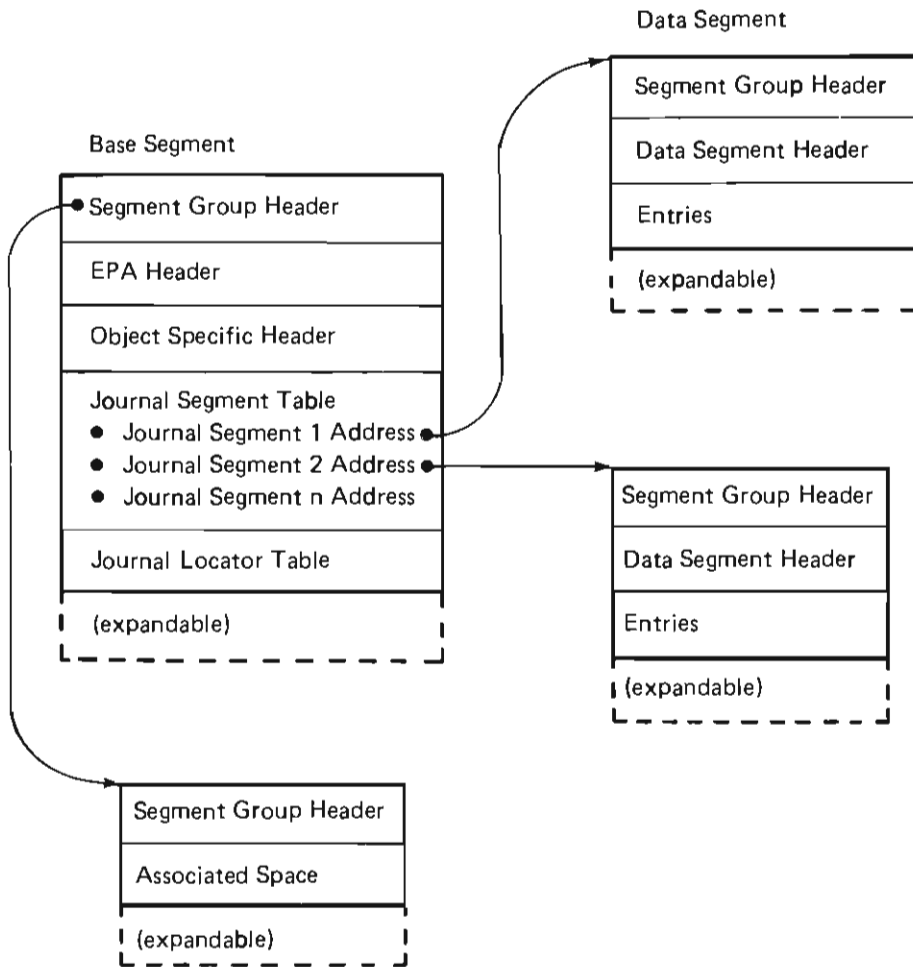


Figure 4-3. Journal Space

One or two journal spaces can be attached to a journal port. If two journal spaces are attached to a journal port, they must be attached at the same time, and their contents must be identical. Once a journal space is detached from the journal port, it cannot be attached to a journal port again and can only be used for find operations.

A journal space can be dumped or reloaded. If the journal space is dumped when it is empty, (not attached to a journal port), it is loaded as empty. If the journal space is dumped when it is attached to a journal port or after it has been detached from the journal port, it is loaded as detached.

Journal management performs the following functions:

- Inserts an entry into the journal space.
- Finds entries on the journal space.
- Performs automatic synchronization during IPL.
- Recovers journaled objects during IPL.
- Performs fix up of journal spaces after the journal spaces are loaded.
- Logs journaled objects on the object recovery list during IPL.

Journal management supports the following System/38 instructions:

- Apply Journaled Changes
- Create Journal Port
- Create Journal Space
- Destroy Journal Port
- Destroy Journal Space
- Journal Data
- Journal Object
- Materialize Journal Port Attributes

- Materialize Journal Space Attributes
- Materialize Journaled Object Attributes
- Materialize Journaled Objects
- Modify Journal Port
- Retrieve Journal Entries

Apply Journaled Changes

#JOAPPLY is invoked as a result of an Apply Journaled Changes instruction. This module applies the recorded changes in a journal space to an object. The recorded changes can be applied in either the forward direction or the backward direction. In the forward direction, the change indicated by the journal entry is applied to the objects in ascending journal sequence number order. In the backward direction, the reverse of the operation specified in the journal space is applied to the objects in descending journal sequence number order.

The Apply Journaled Changes instruction really consists of multiple operations. These operations consist of getting the next journal entry, and making the change to the object.

Create Journal Port

#JOCRTJP is invoked as a result of a Create Journal Port instruction. This module accepts a creation template containing the common create template, plus lengths for the time stamp, process name, user profile name, and program name to be put into the prefix of each entry journaled through this journal port. The object must be created as a permanent object.

Create Journal Space

#JOCRTJS is invoked as a result of a Create Journal Space instruction. This module accepts a creation template containing the common create template, plus a threshold event size. The object must be created as a permanent object.

The journal space is set to the empty status when it is created. This status allows the journal space to be attached to a journal port. In empty status, the journal space has no data segments. The first data segment is created during the execution of the Modify Journal Port instruction that attaches the journal space to a journal port. When the first entry is put into the journal space, the journal entry recognizes that there is not enough space to insert the entry and extends the journal space.

Destroy Journal Port

#JODESJP is invoked as a result of a Destroy Journal Port instruction. This module controls the destruction of a journal port. The following rules control the destruction of a journal port:

- If the journal port is damaged, it can be destroyed with no restrictions.
- If the journal port is not damaged, it cannot be destroyed if it has undamaged journal spaces attached or if there are undamaged objects journaled through the journal port.

Destroy Journal Space

#JODESJS is invoked as a result of a Destroy Journal Space instruction. This module controls the destruction of a journal space. The following rules control the destruction of a journal space:

- If the journal space is attached to a journal port, it may be destroyed only if the journal port is damaged.
- If the journal space is not attached to a journal port, it may be destroyed only if its use count is zero. The use count of an unattached journal space may be nonzero if the Modify Journal Port instruction used to detach the journal space did not complete properly or if the journal space is required by commitment control to perform a decommit.

Note: Even if the journal space is damaged, it may not be destroyed unless it meets the preceding criteria. If a Modify Journal Port instruction fails before it is completed, leaving the use count nonzero, the only way to zero the use count is to perform an IPL.

Journal Data

#JOURDAT is invoked as a result of a Journal Data instruction. In turn, the Journal Data instruction invokes #JOURNAL to provide the facilities for inserting journal entries into the journal space(s) attached to a journal port.

Journal Object

#JOJOB is invoked as a result of a Journal Object instruction. This module initiates the journaling of an object or terminates the journaling of an object. An object is journaled to a journal space through a journal port. The object to be journaled is associated with a journal port with the Journal Object instruction. Starting journaling of an object to a journal port causes an entry to be placed in the journal space(s) attached to the journal port. The Journal Object instruction also removes the association between the object and journal port, stopping recording of changes to that object.

Materialize Journal Port Attributes

#JOMATJP is invoked as a result of a Materialize Journal Port Attributes instruction. This module returns a creation-like template for a journal port. This template consists of the common object creation template and four prefix lengths. The Materialize Journal Port Attributes instruction also returns the number of journal spaces attached to the journal port as well as the pointers to those spaces.

Materialize Journal Space Attributes

#JOMATJS is invoked as a result of a Materialize Journal Space Attributes instruction. This module returns a creation-like template for a journal space. This consists of the common object creation template and the threshold size. The Materialize Journal Space Attributes instruction also returns a pointer to the journal port to which the journal space is attached, and the current state of the entries on the journal space. The current state includes the first and last sequence numbers, attach and detach time, prefix lengths, and the length and sequence number of the longest entry.

Materialize Journalled Object Attributes

#JOMATOA is invoked as a result of a Materialize Journalled Object Attributes instruction. This module returns the journal status for any object including those that cannot be journalled. The instruction returns the object journal attributes, a pointer to the journal port if it is currently being journalled, and the last (current) journal identification.

Materialize Journalled Objects

#JOMATJO is invoked as a result of a Materialize Journalled Object instruction. This module returns a list of objects being journalled through the selected journal port. This list may contain system pointers, object identifiers, and/or journal identifiers.

As the list in the journal port is scanned to return the information, the addresses are checked to make sure the object addressed by the journal port object table also points back to the journal port. The journal port object table may contain entries that do not address journalled objects because the table entry is created before the start of journaling, and is destroyed after the end of journaling. If such an entry is found while materializing the journalled object, the entry is removed from the journal port.

Modify Journal Port

#JOMODJP is invoked as a result of a Modify Journal Port instruction. A journal space must be attached to a journal port before it can become a receiver for journal entries. Similarly, a journal port must have at least one journal space attached before it can be used to journal changes to the objects being journalled through that journal port. #JOMODJP attaches new journal spaces and detaches currently attached journal spaces. #JOMODJP may detach all of the journal space(s) to the journal port. Once a journal space has been detached, it can never be attached to a journal port again.

Retrieve Journal Entries

#JORETEN is invoked as a result of a Retrieve Journal Entries instruction. This module retrieves journal entries from requested journal spaces.

Load/Dump and Suspend

The journal space can be loaded and dumped. A journal space can be loaded again if it is empty or suspended, or if the media version has the same first sequence number and a last sequence number that is at least as large as the storage version. The use count must also be nonzero.

When a journal space is loaded, journal management is called to correct the segment table and other pointers in the journal space object specific header. If the object was attached when it was saved, IPL recovery is run on the journal space to insure that it is in a consistent state.

If the journal space was dumped in an empty state, then it will still be empty when loaded. If the journal space was dumped as either attached or detached, then it will be in the detached state when it is loaded.

A journal space may be dumped at any time. If the journal space is attached when it is dumped, then the dump code seizes the journal to guarantee that the journal is at a block boundary. While the journal is seized, the current insert address and the sizes of the segments are saved. Any changes made to the object after the dump begins will not be reflected when it is loaded.

A journal space can be suspended when its use count is zero. (It is not attached and not needed for IPL synchronization.) When the journal space is suspended, the data segments are destroyed, and the base segment and associated space segment are truncated to one page.

IPL Recovery

If the machine crashes, VMC must, if possible, restore objects to a usable state. Since journaling is a recovery/redundancy function, restoring objects to a usable state is more important than usual.

This portion of the recovery is concerned only with the linkage between journal ports and journal spaces. However, failure to restore those linkages may disallow later recovery operations that require the journal to be as well defined and as complete as possible.

To prevent excessive numbers of writes during run time, only the journal entries are forced. When the machine crashes, it is necessary for the journal recovery function to start at the last known force point in the header and reconstruct the header from the entries.

IPL Synchronization

The purpose of the journaled object synchronization phase of IPL is to assure that the journaled objects are consistent with the journal entries in the journal spaces. Since changes are journaled before any changes are made to the object, it is possible that the object does not contain all changes recorded in the journal space and may need to be brought up to date.

The journal IPL synchronization routine is invoked during every IPL to determine if synchronization operations are required. If synchronization operations are not required, the journaled objects are marked in synchronization with the journal. Otherwise, the necessary operations are performed and then the objects are marked in synchronization with the journal. Following object synchronization, the receiver table (Figure 4-2) is checked to determine if the journal spaces are detached. If the journal spaces are detached, the receiver table is cleared to complete the interrupted Modify Journal Port instruction.

Any time the machine shuts down without writing main storage to nonvolatile storage or not at a boundary (as in a crash), it is possible that some changes to the journaled object may have been recorded on the journal space but the pages actually changed in the journaled object were lost when the machine went down. This routine assures that the journal and journaled object are consistent by assuring that all recorded changes to the journaled object have been written to nonvolatile memory.

Since it is not known which changes were written out to nonvolatile storage and which were lost, IPL synchronization must perform all changes since the last time the journaled object and journal were known to be synchronized.

IPL synchronization needs to interrogate the object table in the journal port header; and, if synchronization is required, it must retrieve journal entries as well as examine and modify the journaled objects. IPL synchronization runs after the initial cleanup of both journal and journaled objects.

DATA AREAS

System-Wide Journal List

Journal management controls a segment called the system-wide journal list (see Figure 4-4) that contains a list of journal ports and journal spaces. The system-wide journal list contains addressability to all journal ports, and to all journal spaces that have a nonzero use count. The system-wide journal list contains 4 bytes for each object, a 3-byte segment identifier and a 1-byte object type. The system-wide journal list is used by #JOINIT to recover journal ports and journal spaces during IPL. If the system-wide journal list segment is damaged, it is recreated, and all segments in the machine are scanned to refill it.

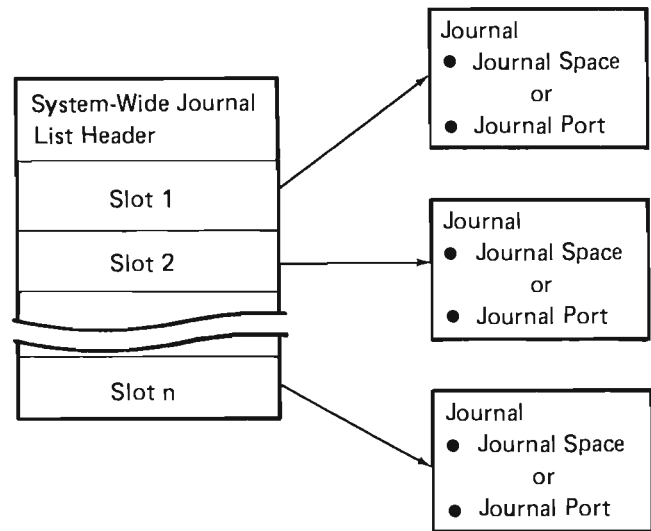


Figure 4-4. System-Wide Journal List

Object Recovery List

The object recovery list contains a list of journal ports and journal spaces that are recovered during IPL.

All journal ports are logged in the object recovery list. The log entry is made by IPL synchronization and, in addition to listing the usual damage bits, it indicates if all objects journaled through the journal port are synchronized.

Attached journal spaces are logged on the object recovery list if they satisfy any of the following conditions:

- Hard damage
- Soft damage
- Suspended
- Unusable
- Past threshold
- Not synchronized
- Journal port is damaged

Any one of these conditions, with the exception of past threshold, prevents any further journaling to the journal space.

STRUCTURE

The following is a list of the modules in journal management and the function that each module performs. The list also shows how the module is invoked.

#JOABTRL Build Transaction List

Function: Builds the transaction list index.

How Invoked: Within this component.

#JOAPLEH Apply Journaled Changes Exception Handler

Function: Terminates or completes the operation of the apply normal or abnormal termination.

How Invoked: Within this component.

#JOAPLYB Apply Change Backward

Function: Applies a single change to a data space in the backward direction.

How Invoked: Within this component.

#JOAPLYF Apply Change Forward

Function: Applies a single change to a data space in the forward direction.

How Invoked: Within this component.

#JOAPPLY Apply Journaled Changes

Function: Applies the recorded changes in a journal space to an object.

How Invoked: Apply Journal Changes instruction.

#JOASELT Select Journal Entries

Function: Checks the journal entry against any selection criteria.

How Invoked: Within this component.

#JOASRSL Sort Select List

Function: Sorts the selection list provided to the Apply Journaled Changes instruction.

How Invoked: Within this component.

#JOATRNB Process Transaction List Backward

Function: Maintains the transaction list when applying changes and/or scanning backward.

How Invoked: Within this component.

#JOATRNF Process Transaction List Forward

Function: Maintains the transaction list when applying changes and/or scanning forward.

How Invoked: Within this component.

#JOAUTRL Unload Transaction List

Function: Removes the entries from the transaction list and returns them to the user space.

How Invoked: Within this component.

#JOCLOSR Close Journal Locator

Function: Releases the list of journal spaces.

How Invoked: Within this component.

#JOCNLEH Cancel Open Block on Journal

Function: Cancels an open block on the journal (exception handler).

How Invoked: From another VMC component.

#JOCRIT Apply Select Criteria

Function: Accepts or rejects the current entry based on the selected criteria.

How Invoked: Within this component.

#JOCRTJP Create Journal Port

Function: Implements the Create Journal Port instruction that creates the journal port system object.

How Invoked: Create Journal Port instruction.

#JOCRTJS Create Journal Space

Function: Implements the Create Journal Space instruction that creates the journal space system object.

How Invoked: Create Journal Space instruction.

#JODESJP Destroy Journal Port

Function: Implements the Destroy Journal Port instruction that destroys the journal port system object.

How Invoked: Destroy Journal Port instruction.

#JODESJS Destroy Journal Space

Function: Implements the Destroy Journal Space instruction that destroys the journal space system object.

How Invoked: Destroy Journal Space instruction.

#JOFIND Find Starting or Adjacent Journal Entry

Function: Finds the starting or adjacent journal entry.

How Invoked: Within this component.

#JOFINDA Find Adjacent Journal Space Entry

Function: Given a current entry, finds the adjacent entry in the selected direction.

How Invoked: Within this component.

#JOFINDQ Find Starting Journal Space Entry by Sequence Number

Function: Finds the starting entry when the user specifies the sequence number as a starting entry.

How Invoked: Within this component.

#JOFINDS Find Starting Journal Space Entry

Function: Finds the starting journal space entry.

How Invoked: Within this component.

#JOFINDT Find Starting Journal Space Entry by Time Stamp

Function: Finds the starting entry when the user specifies the time stamp as a starting entry.

How Invoked: Within this component.

#JOFINEH Journal Find CSEH

Function: Closes the journal locator on an exception.

How Invoked: Other VMC component.

#JOFNDJS Find Starting Journal Space

Function: Finds the starting sequence number or time stamp and selects the journal space containing the starting entry.

How Invoked: Within this component.

#JOINIT Journal Object IPL Recovery

Function: Cleans up the journal objects after a machine crash and puts out the IPL record on each active journal whether the machine crashes or not.

How Invoked: IPL routines.

#JOIRCJP Journal Port Object Recovery

Function: Cleans up a journal port after a machine crash and recovers any active, attached journal spaces using #JOIRCJS. Reconciles twin active journal spaces with #JORECTW.

How Invoked: Within this component.

#JOIRCJS Journal Space Object Recovery

Function: Cleans up a journal space after a machine crash.

How Invoked: Within this component.

#JOISYNC Journal and Object IPL Synchronization

Function: Synchronizes data spaces with journal during IPL.

How Invoked: IPL routines.

#JOJOB Journal Object

Function: Starts and stops journaling of an object.

How Invoked: Journal Object instruction.

#JOMATJO Materialize Journaled Objects

Function: Materializes the list of objects being journaled through a journal port.

How Invoked: Materialize Journaled Objects instruction.

#JOMATJP Materialize Journal Port

Function: Materializes the creation-like template and returns the current attributes of a journal port.

How Invoked: Materialize Journal Port Attributes instruction.

#JOMATJS Materialize Journal Space

Function: Materializes the creation-like template and returns the current attributes of a journal space.

How Invoked: Materialize Journal Space Attributes instruction.

#JOMATOA Materialize Journaled Object Attributes

Function: Materializes the journal attributes for any object.

How Invoked: Materialize Journaled Object Attributes instruction.

#JOMODEH Modify Journal Port Exception Handler

Function: Restores current area journal spaces and clears old area journal spaces.

How Invoked: Within this component.

#JOMODJP Modify Journal Port

Function: Detaches all currently attached journal spaces and optionally attaches new journal spaces and/or resets journal entry sequence numbers.

How Invoked: Modify Journal Port instruction.

#JOOPENR Open Journal Locator

Function: Discovers and seizes explicit or implicit journal space lists.

How Invoked: Within this component.

#JOPREMV Take Entry Out of Journal Port Table

Function: Takes an object entry out of the journal port table.

How Invoked: Within this component and by load/dump.

#JOPUTEH Put in Exception Handler

Function: Backs out of putting an object entry into the journal port entry table when an error occurs.

How Invoked: Within this component and other VMC components.

#JOPUTIN Put an Entry in Journal Port Table

Function: Puts an object entry in a journal port entry table.

How Invoked: Within this component and by load/dump.

#JORECTW Reconcile Twin Journal Spaces

Function: Ensures that active twin journal spaces are identical during IPL recovery.

How Invoked: Within this component.

#JORETEN Retrieve Journal Entries

Function: Implements the Retrieve Journal Entries instruction, which allows the user to selectively retrieve entries from the journal.

How Invoked: Retrieve Journal Entries instruction.

#JOTCHTB Touch Locator Table

Function: Brings the next block of locator table entries and touches the locator entry to detect damage.

How Invoked: Within this component.

#JOTOUCH Touch Current Entry

Function: Perform brings of journal entries and touches current entry to detect damage.

How Invoked: Within this component.

#JOURDAT Journal Data

Function: Implements the Journal Data instruction that allows the user to put entries onto the journal.

How Invoked: Journal Data instruction.

#JOURNAL Journal Entry

Function: Inserts a journal entry, and/or starts or ends a block of journal entries, and/or force the journal.

How Invoked: Other VMC components.

Queue Management

INTRODUCTION

Queue management provides the functions necessary for concurrently executing processes to exchange information as shown in Figure 5-1. A queue is also used for communications between an I/O management task and a process when an I/O request is completed.

A queue provides an object, sharable by processes executing in the system, that can be used to send and receive information. The basic unit of information in a queue is a message. Messages can be sent to (enqueued) and received from (dequeued) a queue. Messages can contain a key used in identifying or sequencing the messages; these are keyed messages.

Messages are processed in move-mode and can be processed in first-in-first-out; last-in-first-out; or keyed in sequence. The basic types of dequeue operations are as follows:

- Dequeue
- Dequeue and set indicator
- Dequeue and branch

A process issuing a dequeue operation (no indicator or branch options specified) is placed in a waiting state if the queue is empty or there are no messages of a particular key. A process issuing a dequeue operation can specify a limit on the length of time it is to wait for a message or the process can wait indefinitely. A message arriving on a queue for which multiple processes are waiting is given only to the first waiting process.

The dequeue and set indicator operation sets an indicator based on whether or not a message was dequeued; then it continues to the next instruction. The dequeue and branch operation allows a process to continue processing at a point that is determined by whether or not a message was dequeued.

Queue management supports the following System/38 instructions:

- Create Queue
- Destroy Queue
- Dequeue Message
- Enqueue Message
- Materialize Queue Attributes
- Materialize Queue Messages

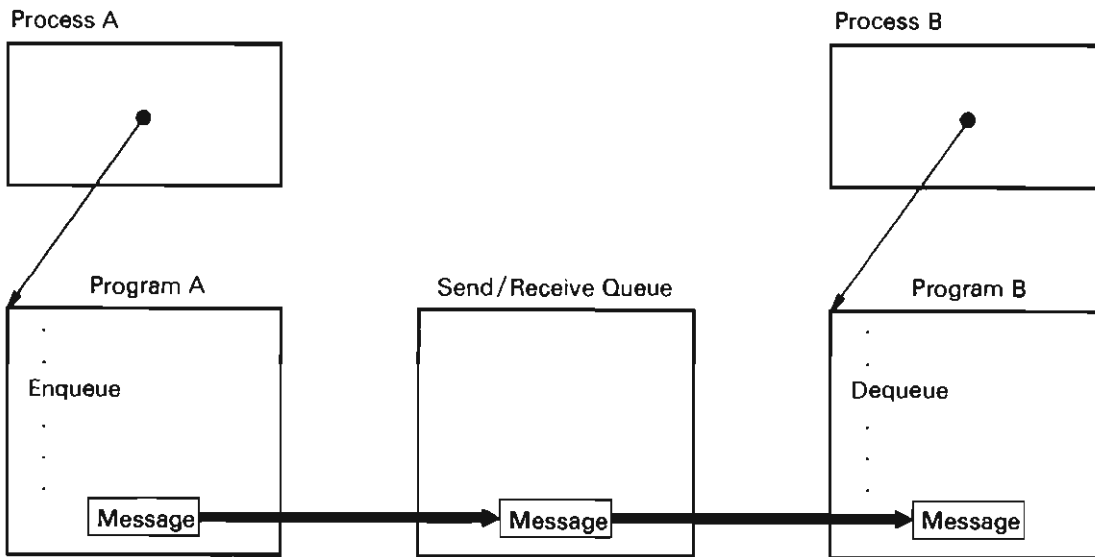


Figure 5-1. Communication Between Processes

Recovery

Component-specific exception handlers (CSEHs) are used in all queue operations to ensure that a queue is not left in an unusable state and that the damage bit is set in the header (if required) before operation terminates.

When a microprogramming exception occurs, the damage bit in the queue header is set. If the machine abnormally terminates and messages remain on a queue, the initial program load (IPL) numbers in the queue and the current IPL number are compared. If the numbers do not match, the integrity of the queue is checked, and the IPL number is updated. Otherwise the damage bit in the queue header is set and the queue instruction operation is terminated.

DATA AREAS

Queues

A queue is created in a segment group. An additional segment group is also allocated if it is a composite object. The queue is designed to function in move mode; that is, all messages enqueued and dequeued are moved to and from preformatted message elements that are part of the queue. The source or target of a message on an enqueue or dequeue operation is designated by a space pointer.

The structure of a queue is shown in Figure 5-2. The encapsulated program architecture (EPA) header is at the beginning of the object, followed by the object specific header. The object specific header consists of two send/receive queues, control fields, and message elements. The first queue is called the response queue; the second is called the available queue. The control fields hold such information as maximum number of messages, number of message elements available, and other queue attributes.

When a queue is initially created, all messages that the queue can contain are formatted and enqueued to the available queue; no messages are enqueued to the response queue.

A bit is set in the last message in the available queue. When this message is dequeued, a send/receive message (SRM) access exception is signaled. A bit is then set in the queue header that signals a queue access exception for an unsuccessful dequeue, and processing continues. An attempt to obtain another message from the available queue causes a queue access exception. If the queue was created with an extendable attribute, the queue is extended when the last message is dequeued.

A destroy message is formatted when a queue is created. When this message is enqueued, an access exception is signaled. This exception causes the destroy functions to be invoked.

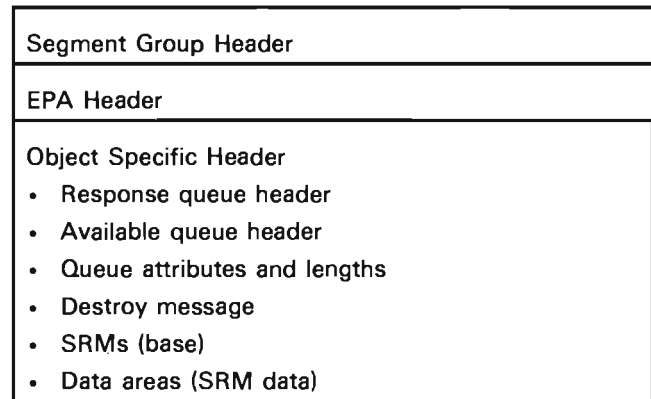


Figure 5-2. Structure of a Queue

Message Elements

Message elements are split into two parts as shown in Figure 5-3. The first part, the base, consists of a 2-byte descriptor, a chain pointer, a key (if present), and a pointer to the other part of the message. The second part, the message, consists of a timestamp and the message text.

The base portion of a message cannot cross a page boundary; any byte of the text portion can cross a page boundary. The message text can contain pointer and character data. If a queue is extended, the new messages are formatted and enqueued to the available queue.

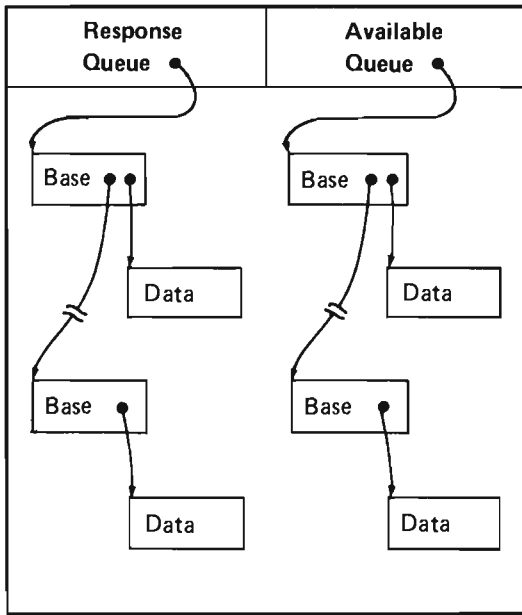


Figure 5-3. Message Elements

STRUCTURE

The following is a list of the modules in queue management and the function that each module performs. The list also shows how the module is invoked.

#PMCRQUE Create Queue

Function: Creates a system object called a queue that is used for interprocess communication or synchronization of interrelated processes.

How Invoked: Create Queue instruction.

#PMDQMSG Dequeue Message

Function: Dequeues a message from the specified queue.

How Invoked: Dequeue Message instruction.

#PMDTYQU Destroy Queue

Function: Destroys the specified queue and all currently enqueued messages.

How Invoked: Destroy Queue instruction.

#PMEQMSG Enqueue Message

Function: Enqueues a message on the specified queue in the specified sequence.

How Invoked: Enqueue Message instruction.

#PMMTQAT Materialize Attributes

Function: Materializes the attributes of the specified queue.

How Invoked: Materialize Queue Attributes instruction.

#PMSRMAC Queue Message Access Exception Handler

Function: Processes send/receive message access exceptions that occur when a monitored message is enqueued or dequeued. A send/receive message access exception initiates a queue extension operation by setting the access exception bit in the queue header. A subsequent dequeue causes a queue access exception to be signaled.

How Invoked: Other vertical microcode (VMC) components.

#PMSRQAC Queue Access Exception Handler

Function: Processes send/receive queue access exceptions that occur during queue processing. A send/receive queue access exception initiates a queue destruction or queue extension operation.

How Invoked: Other VMC components.

#PMMATQM Materialize Queue Messages

Function: Materialize the number of messages on a queue, part of each message, or all of each message and the key.

How Invoked: Materialize Queue Messages Instruction.

Space Object Management

INTRODUCTION

Space object management provides the functions that establish and control the storage areas and space objects used in the execution of machine instruction programs. All space objects are explicitly created using the Create Space instruction.

Created space objects are system objects. Space objects can be extended, truncated, copied, initialized, suspended, and destroyed by System/38 instructions. Space object attributes, such as length and initial value, can be materialized and modified. These operations are subject to the authority that the machine instruction set user has for the object.

Space object management supports the following System/38 instructions:

- Create Space
- Materialize Space Attributes
- Modify Space Attributes
- Destroy Space

Create Space

The Create Space instruction designates an area that is to receive addressability to the created space and a pointer to a template to be used to create the space. The supervisor link (SVL) router invokes the create space instruction processor (#SOCRT). This module performs the space creation operation in the following phases:

- Syntax verification
- Object creation
- Object initialization

During the syntax verification phase, #SOCRT verifies the contents of the creation template for completeness and validity. #SOCRT also checks all authorization and provides proper lock enforcement.

During object creation, #SOCRT performs the actual allocation of the physical storage for the space. The storage is allocated according to the attributes specified in the creation template.

During the object initialization phase, #SOCRT inserts the values and data into physical storage. These values and data are necessary to make the storage into a valid object. #SOCRT then completes the header portion of the object and initializes the data portion with the values requested in the template.

Materialize Space Attributes

The Materialize Space Attributes instruction processor (#SOMAT) is invoked by the SVL router as a result of a Materialize Space Attributes instruction. This module first validates that the input pointer specifies a valid space, and then checks all authorization and provides proper lock enforcement. #SOMAT then materializes the attributes of the space into the specified area. The attributes are materialized in the same format as the creation template.

Modify Space Attributes

The Modify Space Attributes instruction processor (#SOMOD) is invoked by the SVL router as a result of a Modify Space Attributes instruction. This module performs the modification to the space in the following phases:

- Syntax/address validation
- Object modification

In the syntax/address phase, #SOMOD ensures that the input system pointer either addresses a space or addresses an object that contains an associated space with extendable attributes. #SOMOD then checks all authorization, provides proper lock enforcement, and ensures that the input size contains a valid value.

In the object modification phase, #SOMOD increases or decreases the physical storage limit for the space, updates the header information, and initializes the data area with the initial values specified at create time (if the storage was increased).

Destroy Space

The Destroy Space instruction processor (#SODES) is invoked by the SVL router as a result of a Destroy Space instruction. #SODES validates that the input pointer addresses a valid space, checks all authority, and provides proper lock enforcement. #SODES then destroys the contents of the physical storage used for the space and deletes the context entry for that space.

Dump Space Management

Dump Space Management provides a way to dump system objects to the internal storage media and perform simple manipulation of the dump data. The internal storage area that receives the dump data is called a dump space.

The source/sink dump operation dumps system objects into a dump space. The system objects can be reloaded with a source/sink load operation.

The dump data in a dump space can be retrieved from one dump space and loaded into another dump space. The target dump space may exist on the originating system or another system.

Dump Spaces

A dump space is an object that serves as a storage area for a dump of other system objects. As such, it provides an online storage alternative to the commonly used offline storage media (diskettes and tape) for dumps and backup.

A dump space contains a storage area for a contiguous string of 8-bit bytes. The storage area size is variable with a maximum size of roughly 2 gigabytes. The size of a dump space can be specified on creation, implicitly extended by the machine for dump and insert operations, or explicitly reset with a modify operation.

Dump space objects provide storage for dump data only. There is no provision for storage of any other type of data.

Dump Space Functions

Operations on dump spaces as objects are supported by the Create Dump Space, Destroy Dump Space, Materialize Dump Space, and Modify Dump Space instructions. These instructions manipulate the dump space rather than the dump data that is contained in it.

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.

After creation, the dump space can 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 data in a dump space can be manipulated through insert and retrieve operations.

Dump Space Materialization

The Materialize Dump Space instruction is used to materialize the attributes related to a dump space so the current attributes can be determined.

The Materialize System Object instruction is used to materialize common system object attributes to determine their current value.

Dump Space Modification

The Modify Dump Space instruction is used to modify certain attributes related to a dump space. The allocation size of the dump space can be reset to the size of the dump data contained in the dump space.

Dump Space Destruction

The Destroy Dump Space instruction destroys a dump space and frees 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.

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 another dump space. Dump data is initially put 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 reloaded 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 machine and cannot be defined by 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 may not be on the same system as 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 allows the subsequent reloading. A load operation operates on the dump data produced from a previous dump operation to load the system objects contained in the dump space.

The Request I/O instruction is used to perform source/sink load or dump operations on a dump space. A dump operation saves the dump space on a load/dump storage media. A load operation restores the dump space from a load/dump storage media.

Dump Space Data Retrieval

The Retrieve Dump Data instruction can be used to retrieve the 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 for its size and that it is packaged with a small amount of additional data used for verification 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 initial dump space and continuing in ascending order to the end of the dump data.

The format of the dump data to be inserted is undefined other than for its size and that it is packaged with a small amount of additional data used for verification during its insertion. The verification performed on the data is done to ensure that 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 reloaded.

DATA AREAS

Space Object

A space object consists of a single segment group that contains headers and a space of fixed or variable length. Figure 6-1 shows the format of a space object. The space can contain user data and is used for data storage and manipulations.

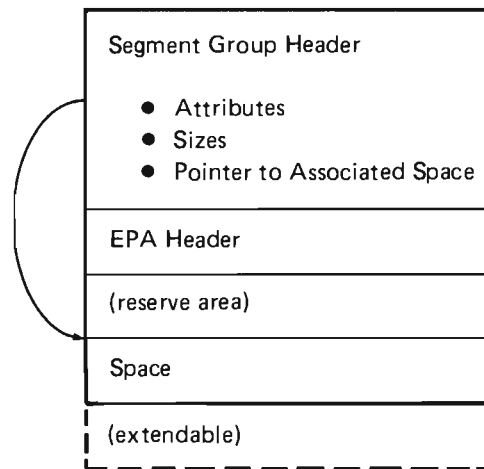


Figure 6-1. Space Object Format

STRUCTURE

The following is a list of the modules in the space object management and the function that each module performs. The list also shows how the module is invoked.

#SOCRT Create Space Instruction Processor

Function: Creates a new space according to the input specifications.

How Invoked: Create Space instruction.

#SOCRTEH Create Space Object
 Component-Specific Exception Handler
 (CSEH)

Function: Processes exceptions that occur during a create space object operation.

How Invoked: Third-level exception handler.

#SODES Destroy Space Instruction Processor

Function: Destroys the specified space object.

How Invoked: Destroy Space instruction.

#SOMAT Materialize Space Attributes Instruction
 Processor

Function: Retrieves and materializes the attributes of the specified space object.

How Invoked: Materialize Space Attributes instruction.

#SOMOD Modify Space Attributes Instruction
 Processor

Function: Modifies the attributes of the specified space object.

How Invoked: Modify Space Attributes instruction,
#AICALLX (process automatic storage area extension),
and #AICRACT (process static storage area extension).

Auxiliary Storage Management

INTRODUCTION

Auxiliary storage management (ASM) performs the following functions:

- Allocates secondary (auxiliary) storage
- Assigns the segment identifiers (SIDs) by which objects and segments are addressed
- Maintains directories that enable 6-byte virtual addresses to be translated into auxiliary storage locations

ASM functions to create, extend, truncate, or destroy segments are invoked by other vertical microcode (VMC) components.

Invoking ASM Functions

ASM functions (create, extend, truncate, and destroy segments) are invoked as a result of Supervisor Link instructions that route control to #SV2DCRT.

#SV2DCRT performs common setup operations, determines whether or not an access group is involved in the operation, and invokes #SMASM to process non-access group requests or #SMAGM to process access group requests. Figure 7-1 shows an overview of access group processing; Figure 7-2 shows an overview of non-access group processing.

Some processing is common to all requests for storage allocation. This processing is performed by #SV2DCRT. This module first determines if either an access group or non-access group function is requested; this is accomplished as follows:

- If a create operation, an explicit parameter identifies the type (permanent, temporary-non-access-group, or temporary-access-group member) of segment that is to be created.
- If an extend, truncate, or destroy operation, the base segment being operated on is provided. Part of the segment address identifies the type (permanent, temporary-non-access-group, or temporary-access-group) of segment.

#SV2DCRT then passes control to either #SMASM or #SMAGM. When that function completes processing, it is possible for events or exceptions to be signaled. An event is signaled if the amount of available auxiliary storage that remains is less than a user-specified limit. An event is also signaled if the number of segment group identifiers remaining for permanent or temporary segments is less than a user specified limit. Exceptions are signaled when the caller specifies that an exception be signaled when an error occurs or when the return code set by the invoking function indicates an error. A second event is signaled if the temporary storage limit for a given process is exceeded.

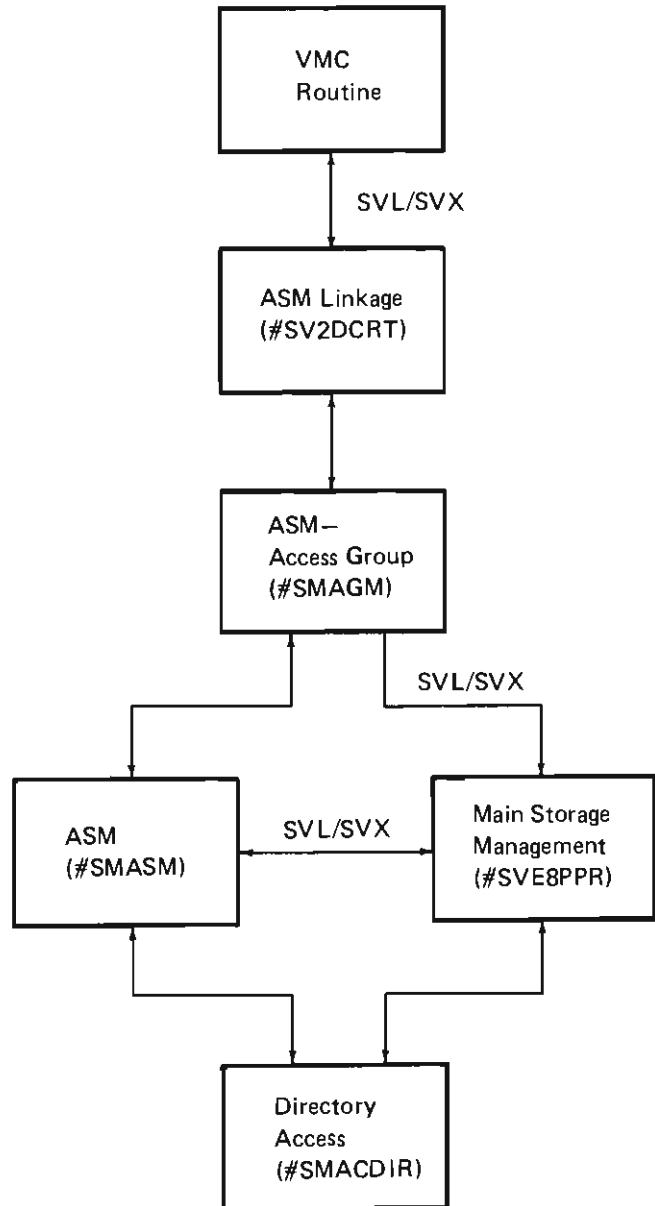


Figure 7-1. ASM Overview of Access Group Processing

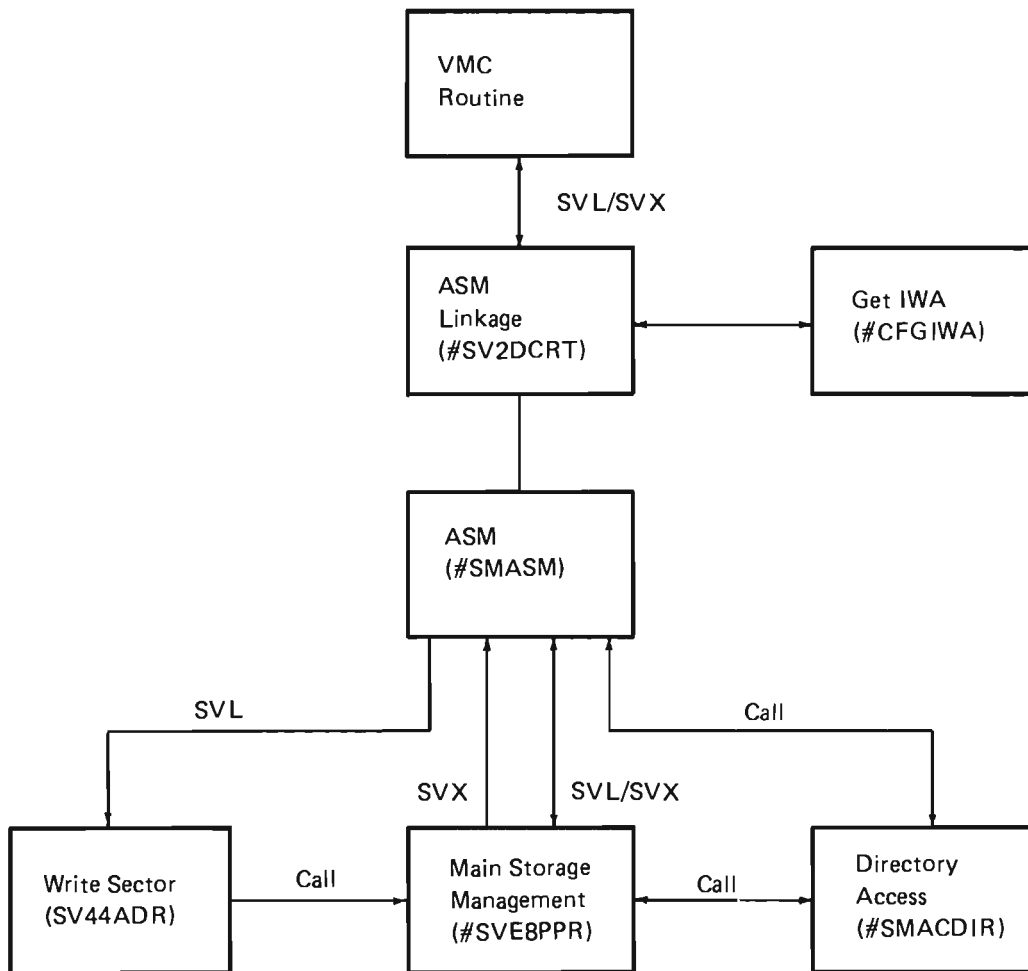


Figure 7-2. ASM Overview of Non-Access Group Processing

Space Accounting

Because auxiliary storage is an important resource, accounting of this resource occurs during ASM operations. User profiles define the space allowed for permanent storage operations; a field in a user process defines the space allowed for temporary and access group segments. These values can be used to limit and control the creation and extension of segments by a process.

If the creation or extension of a permanent segment would cause the user profile auxiliary storage limit to be exceeded, the creation or extension is not performed and a return code is set. If the creation or extension of a temporary or access group segment would cause the process auxiliary storage limit to be exceeded, the creation or extension is performed and an event is signaled to note the occurrence. The process limit is then set to the largest value possible so that the event is not signaled again unless the user changes the process limit and it is exceeded again.

Access Group Processing

Access groups are created, materialized, and destroyed as a result of System/38 instructions. These instructions are processed by resource management. See *Resource Management* in this manual for information concerning creating, materializing, and destroying access groups.

If the requested ASM function involves an access group, #SV2DCRT invokes #SMAGM at one of the following entry points to perform the requested function:

- #SMAGCRT: Create segment in access group
- #SMAGEXT: Extend segment in access group
- #SMAGTRC: Truncate segment in access group
- #SMAGDES: Destroy segment in access group

Create Segment in Access Group

#SMAGCRT first determines if sufficient space in the access group is available to satisfy the request. If enough space is not available, #SMASM is invoked to extend the access group by 32 KB. The check and extension are repeated until enough space is obtained to satisfy the request, up to the 4 MB limit of an access group size. The extension operation is performed by #SMASM as if a VMC routine was extending a temporary, non-access group segment.

After enough space is obtained, a segment number is assigned to the new segment and inserted into the access group member directory along with the segment number of the access group. Space from the access group is then allocated for the new segment and the access group table of contents is updated to specify the storage location of the newly created member pages. The segment group header in the first page of the new segment is then initialized.

Extend Segment in Access Group

#SMAGEXT checks to determine that the segment group to be extended exists, and then determines if sufficient space in the access group is available to satisfy the request. If sufficient space is not available, the access group is extended as with a create segment operation. Once the access group has been extended to a size sufficient to satisfy the request, space from the access group is allocated for the extension and the access group table of contents is updated to specify the storage location of the newly created pages.

Truncate Segment in Access Group

#SMAGTRC checks that the segment to be truncated exists. The segment is truncated by writing null value in the access group table of contents for the pages to be deleted. The access group is then compressed to remove any unused spaces left by deleted pages. A remove request is issued to remove any truncated pages from main storage.

Destroy Segment in Access Group

#SMAGDES performs the destroy segment operation. This operation is similar to a truncate operation except that the entry for the destroyed segment is removed from the access group member directory.

Serialization of Access Group Operations

The access group free space lock is used to serialize most operations. The access group directory lock (applicable to all access groups) is held exclusively when updating the access group member directory or the access group table of contents. The access group directory lock is used to serialize ASM and main storage management (MSM) operations on the same access group.

Non-Access Group Processing

If the requested ASM function does not involve an access group, #SV2DCRT invokes #SMASM at one of the following entry points to perform the requested function:

- #SMSGCRT: Create segment
- #SMSGEXT: Extend segment
- #SMSGTRC: Truncate segment
- #SMSGDES: Destroy segment

Create Segment

#SMSGCRT (entry point in #SMASM) validates the request and determines if sufficient space is available to satisfy the request. If space is not available, #SMSGCRT sets a return code and returns control to the caller.

If sufficient space is available, the next available segment group identification is determined. (If this causes the segment group identification generator to reach hex 3FF000, machine execution is immediately terminated.) The necessary space is then allocated from the free space directory, and auxiliary storage directory entries (ASDEs) are built and inserted into either the permanent or temporary directory.

If the request is for a permanent segment, a page of 0's and the storage management header is written for the first two pages of each allocated extent, except for the first extent. The header is written on the second page only of the first extent. The existence bit in the segment header (page 0) is set to 0 (implying that the segment does not exist). This page must be written to auxiliary storage using the Perform Paging Request instruction. The Perform Paging Request instruction is issued from the process that invoked the ASM function after the create operation is logically complete (for example, the object header is completed and the secondary segments are created). The invoker sets the existence bit to 1 before issuing the Perform Paging Request instruction. Other ASM operations are not serialized during the writing of the headers. See *Serialization (Non-Access Group)* in this section for additional information.

Extend Segment

#SMSGEXT (entry point in #SMASM) validates the size of the requested extension, that the segment exists, and that there is sufficient space available to satisfy the request. If the request is not valid, #SMSGEXT sets a return code and returns control to the requestor. If the request is valid, #SMSGEXT allocates the space and writes the headers for each new extent as with a create segment operation.

Destroy Segment

#SMSGDES (entry point in #SMASM) first determines if the segment exists. If not, #SMSGDES sets a return code and returns control to the requestor.

For temporary segments, directory entries are removed from the temporary directory and the extents of free space are returned to the free space directory (they are combined with adjacent free extents to form larger blocks where possible).

Next, #SMPLAD (entry point in #SMACDIR) is invoked to purge the affected addresses from the lookaside directory. Then a remove perform paging request is issued. This request invokes #SVE8PPR to delete addressability to any pages of the destroyed segment that are currently in, or being paged from or to main storage.

For permanent segments, the destruction is logically performed a single ASDE at a time. (An ASDE can contain from one to four extents.) This is accomplished as follows:

- The ASDE is removed from the permanent directory.
- #SMPLAD is invoked and a remove Perform Paging Request instruction is executed to delete addressability to the pages addressed by the ASDE.
- MSM is invoked to write a free space header to the first two pages of each extent addressed by the ASDE and the extent is returned to the free space directory.

The writing of the free space header is performed before the extent is returned to the free space directory because the free space lock is released before the write operation.

Truncate Segment

The truncate operation is performed by #SMSGTRC (an entry point in #SMASM). This operation is similar to the destroy operation with the following exceptions:

- The input size is checked to ensure that it is less than the current segment size.
- The extents are freed proceeding from the last directory entry to the first and only until the target address is reached.
- The free space lock is not released.
- If the target address is within an extent (not on an extent boundary), that extent is truncated 1/2, 1/4, 1/8, . . ., 1/32 768 of the original size as appropriate.

Serialization (Non-Access Group)

The free space lock is held throughout most ASM processing. This lock serializes the free space directory and the work areas and fields used by ASM in the storage management vector table (SMVT). When the permanent and temporary directories are accessed, the corresponding directory lock is held to serialize with MSM operations. The MSM lock is held by #SMPLAD when the lookaside directory is purged, and both MSM and truncate locks are held when a remove perform paging request is processed by MSM. The free space lock is not held when the headers are written for permanent segment create, extend, or destroy operations but the lock is held for all write operations when a permanent segment is truncated.

ASM Locks

The following locks are used for storage management operations. These locks are send/receive counters in the SMVT.

All locks, except as noted, are exclusive.

- **Access Group Free Space Lock:** This lock serializes access group processing.
- **Free Space Lock:** This lock (held throughout most of non-access group processing) serializes the free space directory, SID generator, and various other ASM areas in the SMVT.
- **Access Group Directory Lock (exclusive and shared):** This lock synchronizes ASM and MSM operations that involve access groups. When ASM updates either the access group member directory or an access group table of contents, ASM holds the lock exclusively. When MSM interrogates either the directory or table of contents, MSM holds the lock shared.
- **Permanent and Temporary Auxiliary Storage Directory Locks:** These locks are held while the corresponding directory is examined or updated.

Auxiliary Storage Initialization

A module (#SMASI) is loaded and invoked by the service monitor. This module performs the following functions:

- Moves the factory defect maps to cylinder 358 (62PC only).
- Moves alternate sectors to cylinder 358 (62PC only).
- Writes 0's to the header and data areas of each sector.
- Allocates two defect-free areas on drive 1 for segments loaded during horizontal microcode (HMC) initial microprogram load (IMPL).
- Creates storage management directories on drive 1.
- Allocates the prebuilt segments.
- Writes the SMVT onto drive 1. This SMVT contains the directory valid bit, the segment identifier generator, the static directory, the free space values, and the auxiliary device configuration record.
- Enters free space for added actuators and updates the number of actuators used by storage management in the SMVT.

Because the machine configuration record on the disk is just another pageable segment, storage management startup needs a different record than the machine configuration of the number and types of actuators. This record is stored in the SMVT. The relevant portion of the SMVT is updated by auxiliary storage initialization. If the machine configuration record update on disk followed by a link/load is not performed after drives are added to the system, the subsequent IMPL will fail in storage management startup.

Storage Management Shutdown

Shutdown occurs whenever the machine is brought down in a controlled state. Shutdown must always be run after an IMPL operation has proceeded past main storage initialization if directory recovery is to be avoided.

Shutdown (#SMSHTDN) is normally invoked as a result of the Terminate Machine Processing instruction, though errors internal to VMC not related to storage management can also cause #SMSHTDN to be invoked as part of system failure shutdown procedure.

Shutdown obtains the access group free space lock and the free space lock. This inhibits any create or destroy activity. Shutdown then cycles through the temporary directory, reclaiming all extent descriptors and returning the space to the free space directory. Shutdown then cycles through the primary directory, writing all changed pages to auxiliary storage. #SMSHTDN sets the directory good bit, writes the SMVT checkpoint page along with the directory good bit and other related areas, and returns to the termination routine. Note that the locks obtained during shutdown are still held after the exit preventing further ASM activity.

Some diagnostic routines (display/alter in particular) running under service monitor 1 can examine the content of permanent and temporary storage as it existed when shutdown was invoked. On the next IMPL operation, the destruction of temporary space is completed when #SMMSIT empties the temporary directory.

Directory Recovery

Storage management directories are used to map virtual addresses to auxiliary storage locations. Each sector on auxiliary storage is defined by an 8-byte header that contains the virtual address associated with a page. The header also contains the size of the extent (contiguous block of auxiliary storage allocated to a segment), indicators of valid pointers on the page, and flags associated with the segment. If it is determined that the directory is unusable, a new directory can be constructed by reading each sector on auxiliary storage.

In general, the storage management functions that perform directory recovery operations interact as follows:

- ASM (#SMASM) writes the headers during operations on permanent segments so that the existence, location, and size of segments can be determined. For create requests, the first page of the first extent is not written. When the invoking routine has completed the create operation, it sets the existence bit in the segment header to on and writes the page to auxiliary storage. ASM also maintains headers with a preassigned virtual address on the first page of large extents of unallocated space.
- MSM (#SVE8PPR) always preserves record headers during write operations.
- A bit in the SMVT (#SMSMVT) indicates if the directories are valid. At IMPL, the bit is set to off (directories invalid) and the SMVT is written to auxiliary storage.
- The storage management shutdown routine (#SMSHTDN) is invoked when the machine terminates (normally or abnormally). If shutdown is successfully accomplished, #SMSHTDN writes all changed directory pages in main storage to auxiliary storage, sets the directory valid bit on, and writes the SMVT to auxiliary storage.
- During IMPL, #SMMSIT checks the directory valid bit in the SMVT (which is loaded with the nucleus). If valid, IMPL proceeds. If not valid, the directory recovery program is invoked. (The system operator is informed that directory recovery is in progress.) The directory recovery program rebuilds the directories and IMPL proceeds.

The directory recovery program recovers only the free space directory and the permanent directory. The free space, permanent, temporary, and access group member directories are first reset to empty. The recovery then occurs in two passes. The first pass reads auxiliary storage, constructing immediately reclaimable free space and permanent extent candidates. The second pass examines these candidates and determines which are permanent segments and which are destroyed (thus free space).

Because segments are allocated in contiguous extents whose relative record number is on the same power-of-two boundary as their size, reading from relative record zero to the end of auxiliary storage detects each valid extent. The beginning of an extent is found whenever the virtual address in the header is permanent and the relative record number is an exact multiple of the extent size in the header. Such a valid extent is a candidate for being part of a permanent object. It is stored in a directory and the remaining headers in the extent are bypassed. A large extent of free space is found whenever the virtual address in the header is the preassigned delimiter for large blocks of unallocated space and the relative record number is an exact multiple of the extent size in the header. Such an extent is returned to free space and the remaining headers in the extent are bypassed. Sectors with other headers (zeroed headers, temporary headers, or permanent headers on wrong boundary) are combined into free space extents and returned to the free space directory. When all headers of all devices have been processed, the directory of candidate extents is scanned to reconstruct permanent segments.

Directory recovery requires that all headers be read. If a permanent read error occurs, the header for that sector cannot be accessed. The next sector on disk is read. If that sector is the second page of a permanent extent, its header is used to reconstruct the header of the failing sector. If the second sector is also unreadable, 0's are provided for the header of the original failing sector. In this case, if the original failing sector had been the first page of a permanent extent, the extent is lost. Otherwise, no visible effect occurs. However, if at some later date the page is referenced during normal operations, a permanent read error can occur.

Permanent Directory

The permanent directory is a machine index that contains 11-, 16-, 21-, and 26-byte entries called auxiliary storage directory entries (ASDEs). The ASDEs map the disk addresses assigned to all permanent segments. ASDEs consist of the first virtual address mapped, followed by one through four extent descriptors. The permanent directory is initialized when VMC is installed and is updated by ASM operations. The format of the permanent directory is shown in Figure 7-4.

Temporary Directory

The format of the temporary directory is the same as the permanent directory and is used to map addresses of temporary segments. The format of the temporary directory is shown in Figure 7-4.

Base Segment Identifier	First Page Identifier	Flags	(reserved)	1st Extent Descriptor	...	4th Extent Descriptor
Base Segment Identifier	First Page Identifier	Flags	(reserved)	1st Extent Descriptor	...	4th Extent Descriptor

Figure 7-4. Permanent and Temporary Directories

Access Group Member Directory

This directory maps the segments in an access group to the actual access group. This enables MSM during a page fault to access an object in an access group when that access group is not in main storage. The contents of the access group member directory is shown in Figure 7-5.

Virtual Address of Segment X	Virtual Address of Access Group 1	Size of Segment X
Virtual Address of Segment Y	Virtual Address of Access Group 1	Size of Segment Y

Figure 7-5. Access Group Member Directory

Access Group Table of Contents

The access group table of contents describes the contents of an access group in a manner that enables MSM to operate on either the entire access group or its individual pages. The table consists of 8-byte entries; each entry contains the virtual address of a page of an object in the access group and the status bits for that page. The table of contents also contains auxiliary storage information that is used to map the virtual address of a page in the access group to its auxiliary storage location and special indicators. The format of the access group table of contents is shown in Figure 7-6.

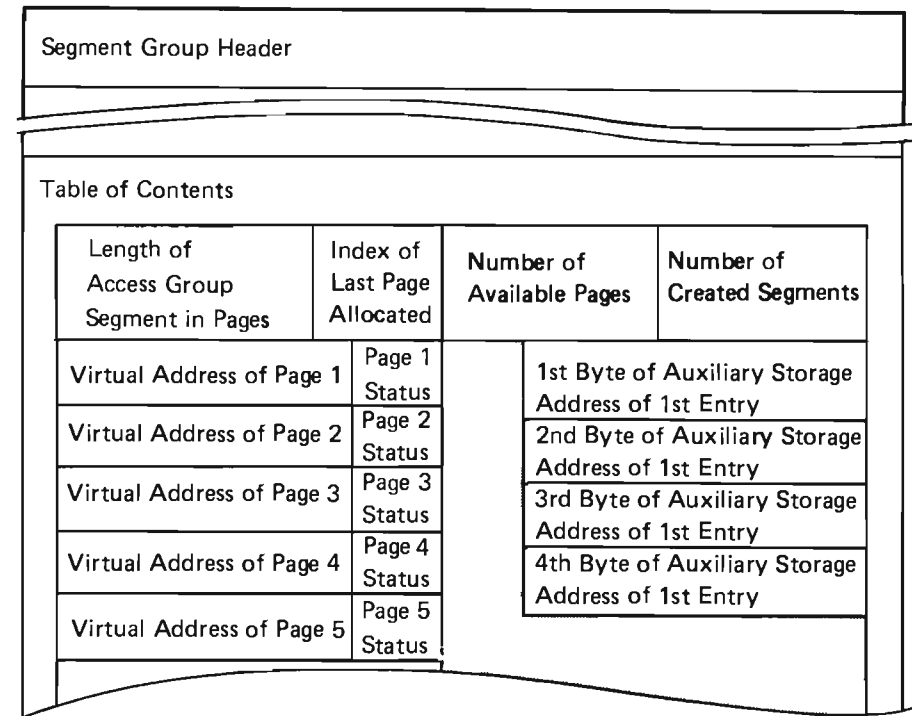


Figure 7-6. Access Group Table of Contents

Storage Management Vector Table

This control block is compiled as a nucleus module (#SMSMVT) and contains the following information:

- Segment identification generators and segment extender
- Storage management locks
- Device characteristics and free space information
- Work areas for ASM
- Index control blocks for directory operations
- Storage management system-wide statistics
- Main storage pool headers

Some of the information in this block is saved on auxiliary storage during shutdown and at certain other critical points.

Sector Headers

All sectors on disk are 520 bytes long. The first 8 bytes form the header which contains the virtual address of the page and associated page and segment information. The sector header is used for the following:

- Allows pages to be made self-defining so that directories can be recovered.
- Enables MSM to determine whether a page has been previously referenced. If, after a page is transferred to main storage, the header does not match the virtual address being read, MSM assumes that this is the first reference to the page and sets the page to zero.

The header also contains information on the location of machine interface pointers within the page.

The header is read or written with every MSM disk operation. For write operations, the information in the header is obtained by MSM from the primary directory element and from the page itself (tags) so it is not necessary to access the directory to construct the header. The following shows the format of the sector header:

Virtual Page Address	Indicators	Reserved	Pointer to First Machine Interface Pointer in Page
----------------------	------------	----------	--

STRUCTURE

The following is a list of the modules in auxiliary storage management and the function that each module performs. The list also shows how the module is invoked.

#SMAGM Auxiliary Storage Management for Access Groups

Function: Creates, extends, truncates, and destroys segments in access groups.

How Invoked: Within this component.

#SMASA Alternate Sector Assignment

Function: Moves factory defect record to track 358, moves alternates to track 358, and writes 0's in tracks 0 through 357.

How Invoked: Auxiliary storage initialization.

#SMASAS Auxiliary Storage Initialization Directory Build

Function: Builds the auxiliary storage directories during auxiliary storage initialization.

How Invoked: Auxiliary storage initialization.

#SMASI Auxiliary Storage Initialization

Function: Moves defect record and alternates to track 358, writes 0's in sectors 0 through 357, and assigns the segment identifiers on drive 1. When initialized, adds the free space for all new drives.

How Invoked: Service Monitor.

#SMASIML Auxiliary Storage Initialization Message Library

Function: Contains the display and message library for auxiliary storage initialization.

How Invoked: Not applicable.

#SMASITD Task Dispatching Element for #SMASI

Function: Contains the Task Dispatching Element for #SMASI.

How Invoked: Not applicable.

#SMASM Auxiliary Storage Management

Function: Performs the create, extend, truncate, and destroy functions for ASM.

How Invoked: Within this component.

#SMEXDIR Extend Storage Management Directories

Function: Extends the segments containing the permanent, temporary, free space, and access groups directories.

How Invoked: Within this component.

#SMSGEX Signal Exception from ASM

Function: Signals an exception for an ASM detected return code.

How Invoked: Within this component.

#SMTHEV Threshold Reached

Function: Signals an event when the ASM threshold limit has either been reached or exceeded.

How Invoked: Within this component.

#SV2DCRT ASM Link Routine

Function: Provides the common linkage to ASM functions.

How Invoked: VMC through an explicit supervisor linkage (SVL).

Main Storage Management

INTRODUCTION

Main storage management (MSM) performs the following paging functions:

- Places pages in main storage when necessary to execute an instruction or perform I/O operations
- Performs specialized paging operations to improve performance
- Places pages from main storage back into auxiliary storage when required
- Manages the pages in main storage

In addition to the preceding paging functions, MSM performs certain initialization, shutdown, and recovery operations.

MSM Paging Function

The paging function (#SVE8PPR) of MSM is invoked by other VMC components as a result of a page fault or a Perform Paging Request instruction. A page fault is an exception that occurs when a task uses a virtual address to access a segment and the page corresponding to that segment is not validly located in main storage. Page faults are converted to a Perform Paging Request instruction (bring, synchronous, no pin) by #SMPFEXH.

A Perform Paging Request instruction is a request to MSM to perform one of the following functions:

- Bring: Bring pages into main storage.
- Exchange Bring: Bring pages into main storage frames that were previously allocated to a specified range of virtual addresses. (If the virtual addresses specified cannot be used, new page frames are allocated.)
- Write: Write changed pages to auxiliary storage.

- Clear: Set pages to binary zeros.
- Exchange Clear: Set pages to binary zeros and, if possible, allocate the pages from a specified range of virtual addresses. (If the virtual addresses specified cannot be used, new page frames are allocated.)
- Remove: Remove the specified pages in main storage without writing them to auxiliary storage.
- Purge: Write changed pages to auxiliary storage and make the pages eligible for reassignment.
- Bring Access Group: Bring active pages of an access group into main storage.
- Purge Access Group: Write all changed pages of an access group to auxiliary storage.

A Perform Paging Request instruction results in an implicit Supervisor Link instruction with three operands. The first operand designates the first page to be operated on, the second operand designates the last page to be operated on. The third operand is contained in byte register 15. This operand contains the Perform Paging Request instruction function requested of #SVE8PPR. In addition to the Perform Paging Request instruction function, the following options can be requested in the third operand:

- Pin: Increases the pin count for pages brought into main storage. (Option for bring and clear.)
- Unpin: Decreases the pin count before the Perform Paging Request instruction is performed. (Option for write, purge, and remove.)
- Synchronous: Allow the requesting task to wait for completion of the requested task. (Option for write, bring, and purge; implied for remove and clear.)
- Asynchronous: Allow the requesting task to continue execution without waiting for completion of the requested operation. (Option for bring and purge; not valid for remove and clear.)

The following functions are performed within #SVE8PPR but are invoked from an interface other than a page fault or a Perform Paging Request instruction:

- Read sector: Read the specified relative sector into the specified page
- Write sector: Write the specified page to the specified sector
- Allocate page frame: Allocate a page frame with a virtual address that is the same as the real address

MSM also provides the following functions:

- Deallocate page frame: Deallocate a virtual-equals-real frame of main storage
- Alter storage pool: Modify the storage pool

Paging Function Tasks

Figure 8-1 shows the structure of the tasks associated with a paging request. These tasks are described as follows:

- User task: A user process, a nonstorage management VMC task, or a nonstorage management operational unit (OU) task.
- Asynchronous I/O task: A task that performs some I/O request and allows the user task to resume execution before the I/O operation is complete.
- Page out task (not shown in the figure): This task writes to auxiliary storage pages that have been modified in main storage and makes these pages eligible for replacement.
- Storage management OU tasks: These tasks provide the interface to the paging devices.

Communications among these tasks are accomplished through paging request elements (PREs), OU queues, and asynchronous paging request queue. The following describes the sequence in the task structure. The step numbers in the description correspond to the numbers in Figure 8-1.

Note: The available PRE queue is actually the available CRE queue (ACQ) that storage management shares with the horizontal microcode (HMC) SVL mechanism. Before a PRE is returned to the ACQ, MSM reformats the PRE as an available CRE.

1. A Perform Paging Request instruction or a page fault occurs, and the user task obtains a PRE from the available PRE queue (the ACQ).
 - a. If the request is asynchronous, the PRE is initialized and sent to the asynchronous I/O input queue and the user task performs a Supervisor Exit instruction.
2. The first operational program is built and the PRE is sent to the OU task queue.
3. The OU task receives the PRE that includes the operational program. (The PRE appears as a standard operation request element (ORE) to the OU task.)
4. The operational program is executed and the PRE is returned to the user task or the asynchronous I/O task.
5. The PRE is received by the appropriate task.
6. If the request is not complete, the next operational program is built, the PRE is sent to the OU queue, and the sequence is repeated beginning with step 3.
7. If the request is complete, the PRE is returned to the available PRE queue and processing completes. The user task performs a Supervisor Exit instruction and the asynchronous I/O task waits for further requests.

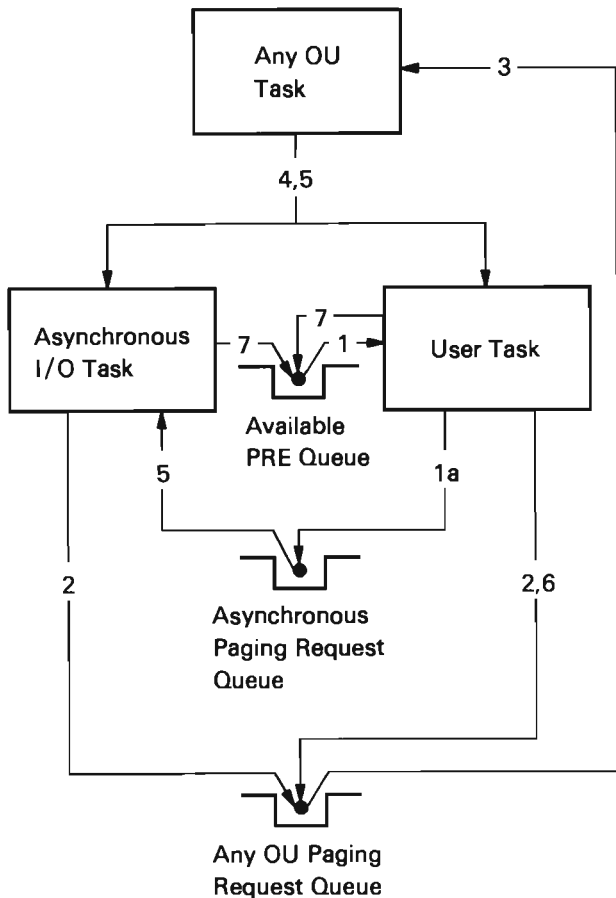


Figure 8-1. Storage Management and Tasks

PRE Processing

The PRE is used to pass information among the tasks involved with a paging request. The PRE contains the last virtual address of the request, the length (minus 1), an ORE, the PRE request code, the storage pool number, and the owning task dispatching element (TDE). See *Data Areas* in this section for additional information concerning the contents of the PRE.

In the case of a page fault, the page fault exception handler (entry point #SMPFEXH in #SVE8PPR) sets up a PRE with the last virtual address (page bounded) and the PRE request code to page fault (synchronous, no pin). The length is not defined at this time. After the storage management directory is accessed, the length is set to one or more pages and the PRE request code is set to read (synchronous, no pin). The pool number is set either to the pool number specified in the TDE of the requestor or to the default storage pool (pool 1).

In the case of a Perform Paging Request instruction, the Perform Paging Request instruction handler (#SVE8PPR) sets up a PRE with the last virtual address equal to the page boundary of the second operand. The length is set to the difference between the first and second operands. The PRE request code is set according to the operation requested in the third Perform Paging Request instruction operand.

As the request proceeds, the pages are processed (singly or in groups) beginning with the last virtual address proceeding backwards to the first virtual address. When processing for a page or group of pages completes, the length is decremented. The original request becomes a new request for the remaining pages of the request.

When the length is 0 (PRELNTH equals -1), all pages have been processed. The PRE and any extensions are returned and control is returned to the user via a Supervisor Exit instruction.

Mainline Processing

Paging request processing for page faults and Perform Paging Request instructions is shown in Figure 8-2. VMC routines invoke the mainline module (#SVE8PPR) for a page request (through the Supervisor Link instruction router) or a page fault (through the first-level exception handler). The first-level exception handler invokes the mainline module at entry point #SMPFEXH.

For bring (including page fault) and clear request, #SVE8PPR invokes #SMACDIR to determine the relative record number (auxiliary storage address) of the requested address(es) and then allocates page frames in main storage. Requests needing I/O operations are then sent to the appropriate OU tasks. The OU task returns the request to #SVE8PPR.

If an I/O error was encountered, #SMERP is invoked under the user task. If any problems (such as a segment not found or uncorrectable I/O error) were encountered, #SVE8PPR invokes #CFSLEH to signal the exception. If no errors were encountered, #SVE8PPR returns control to the user.

The page out task (#SMPOT) is used to write changed pages to auxiliary storage. This task selects the pages to be written to auxiliary storage based on information accumulated by user tasks during page allocation. The pages are written using a Perform Paging Request instruction (purge). This task is initiated asynchronously during page allocation.

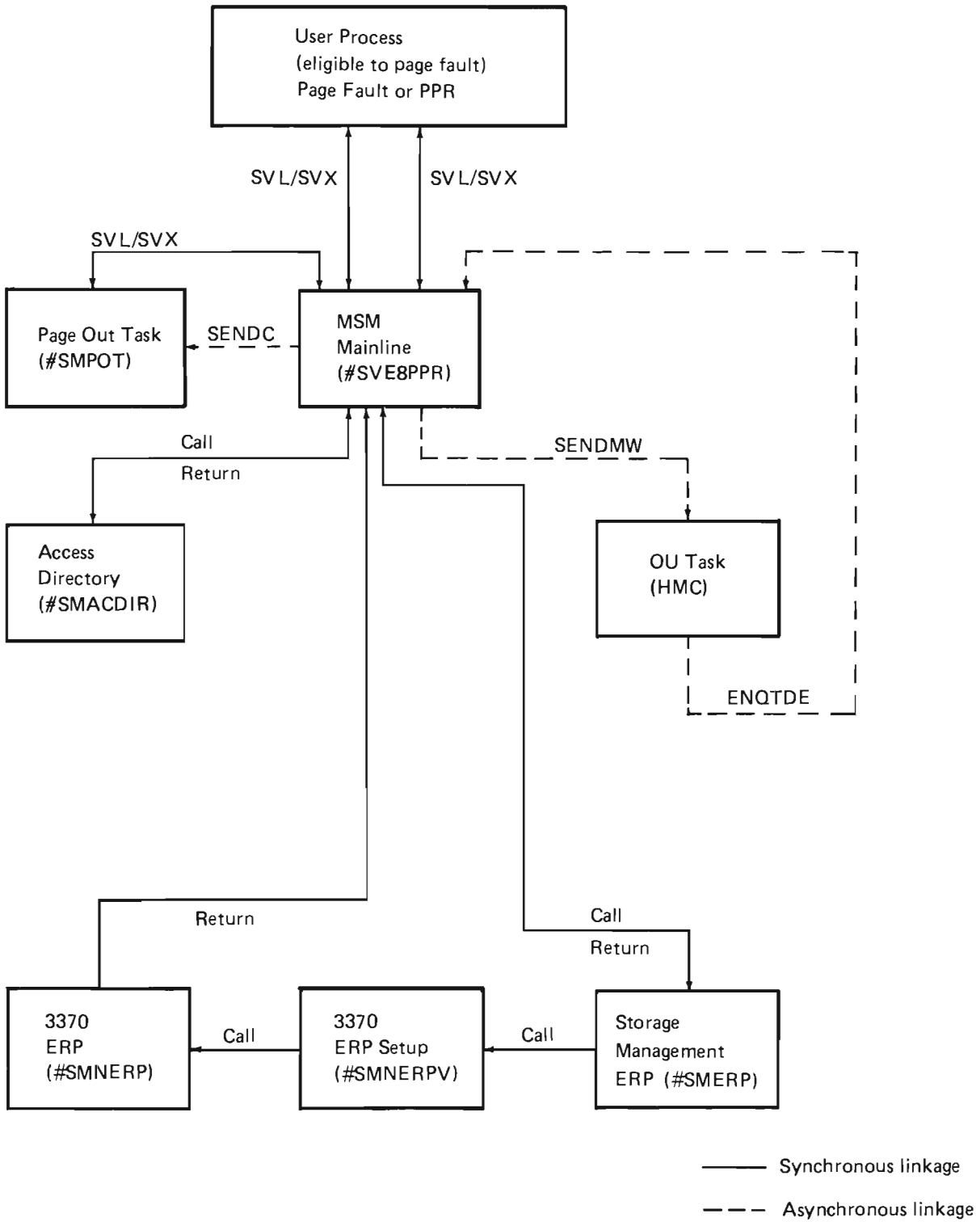


Figure 8-2. Page Fault and PPR Processing

Directory Lookup

For any clear or read request (including a page fault), the disk addresses of the pages are determined by examining the storage management directories. For purge and write requests, this is not required because the auxiliary storage addresses are maintained in the primary directory element (PDE) extension for each page.

Directory lookup is performed by #SMACDIR. If an access group is not involved, and the faulting address or the address specified in the Perform Paging Request instruction is not in main storage, #SMACDIR searches the static directory for the address. If the page is not found in the static directory, the lookaside directory is searched, and if the page is still not located, the temporary or permanent directory is searched. If the address is found in the temporary or permanent directory, the entry is put into the lookaside directory. Access to the lookaside and temporary or permanent directories is serialized by storage management locks as described in this section.

If an access group is involved, the access group member directory is searched to determine what access group contains the address. The access group table of contents for that access group is searched.

If the requested virtual address(es) is not described in any of the directories, an exception is signaled.

#SMACDIR also determines how many pages to read and in some cases, the storage pool to receive them. Page faults normally result in a single page bring. If the block transfer attribute was specified when the segment was created, auxiliary storage management (ASM) has set the block transfer indicator in the auxiliary storage directory entry (ASDE). #SMACDIR interprets this bit as meaning that a block (depending on boundary and extent size restrictions) of up to eight pages containing the faulting page is to be transferred.

Page Replacement

Main storage is partitioned into one or more storage pools. Associated with each pool are two queues, a search queue and a change queue. The search queue is used to locate the next available page. The change queue is used to write the changed pages to auxiliary storage. The page replacement algorithm operates on a pool, rather than machine-wide basis. On all page faults and most bring and clear requests, the PRE contains a pool identifier that identifies the pool of the executing process. The identified storage pool is searched to select a page.

The page replacement algorithm searches from the original first page to the last page in the search queue of a pool until a page is located or the end of the queue is reached. If the end of the queue is reached, a return is issued and the pool is not tried again. This failure situation is described under *Page Out Task* in this section.

The page replacement algorithm examines the following attributes to determine if a page is eligible for replacement:

- Reference bit: This bit indicates if a page has been referenced since it was read into main storage or since it was last examined for replacement. (MSM resets this bit upon examination.)
- Change bit: This bit indicates if a page in main storage differs from the corresponding page on auxiliary storage. (Both the reference and change bits are set by execution of instructions and I/O operations.)
- Pin indicator: This byte indicates if a page can be invalidated. (Storage management normally sets this byte for other VMC routines; internal I/O operations and instructions involving interrupted send/receive queue (SRQ) operations set this byte directly.)
- Secondary reference bit: This bit indicates if the reference bit was on the last time it was examined. (MSM sets or resets this bit.)
- Storage management pinned: This byte in the PDE is used by storage management to indicate special conditions. All special conditions pin the page but do not prevent invalidation of the page. MSM resets any bits set to 1.
- Virtual address equals real address: This condition indicates an unallocated page. This normally occurs for pages or a segment that was recently destroyed. (Nucleus pages are not accessible to the algorithm.)

If a page is unreferenced two times in succession, unchanged, not pinned, or virtual=equals=real and not pinned, the page can be allocated to a new request. If the page is referenced or pinned, it is moved to the end of the search queue. If a page is changed (unreferenced and unpinned), it is moved to the change queue. Figure 8-3 shows the frame attributes and the actions that can be taken for each attribute.

A dash on the chart in Figure 8-3 means a given frame attribute is irrelevant. Reading the chart from top to bottom tells a set of frame attributes under which the action described above occurs. For example, a pinned page in the first column, regardless of its other attributes, is passed over.

Attributes	Action							
	Pass Over ¹			Allocate ²			Transfer to Change Queue ³	
Pinned or storage management	Yes	No	No	No	No	No	No	No
Virtual equals real	-	No	No	Yes	No	No	No	No
Referenced	-	Yes	No	-	No	No	No	No
Changed	-	-	-	-	No	No	Yes	Yes
Referenced last time	-	-	Yes	-	No	-	No	-
Data base page	-	-	No	-	-	Yes	-	Yes

¹Do not allocate, do not transfer to the change queue.
²Allocate the page for the new request.
³Transfer the page to the change queue.

Figure 8-3. Replacement Actions

Page Out Task

The page out task performs the following functions:

- Writes changed pages to auxiliary storage when the system is not busy
- Writes pages to auxiliary storage when a pool has pages on the change queue that are eligible for replacement

When not in the process of writing pages to auxiliary storage, the page out task waits on a send/receive counter called the threshold counter. Each time the page replacement algorithm puts a page on the change queue, it increments the threshold count. When the threshold count exceeds the specified threshold value, a send count is issued to the threshold counter to restart the page out task.

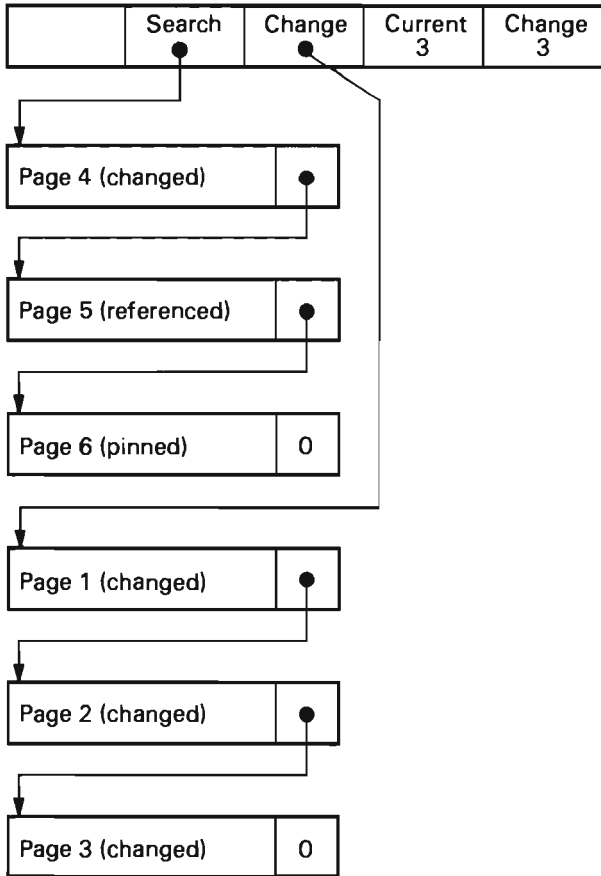
A task enters pool wait when the task attempts to allocate a page for a read or clear operation and the end of the original search queue is reached (all pages passed over or put on the change queue). The task enqueues its PRE to the pool wait queue. The task then sends count to the page out task queue and issues a keyed receive on the pool wait queue. The page out task is restarted.

When the page out task is restarted, it attempts to write up to five pages from the change queue of each pool and transfer the pages to the search queue. Each time the page out task is restarted the threshold count is decremented.

After up to five pages per pool have been written, the page out task restarts any user tasks that are waiting on the pool wait queue. The user task may again attempt to allocate a page.

Figure 8-4 shows an example of the page out function. In the example, a user task needs a page frame and searches its storage pool. After searching the entire pool and not finding an available frame, the task signals the page out task and enters a wait state. The page out task writes all pages to the change queues and restarts all user tasks that are waiting on the pool wait queue.

Pool Before Search



Pool After Search (before page out)

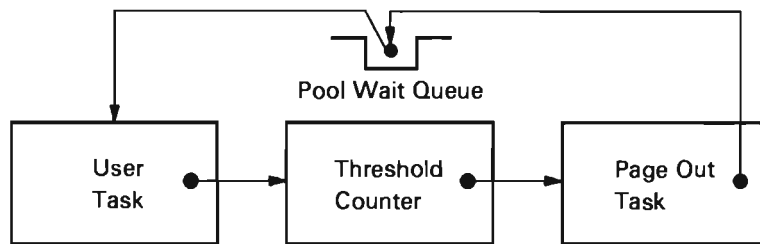
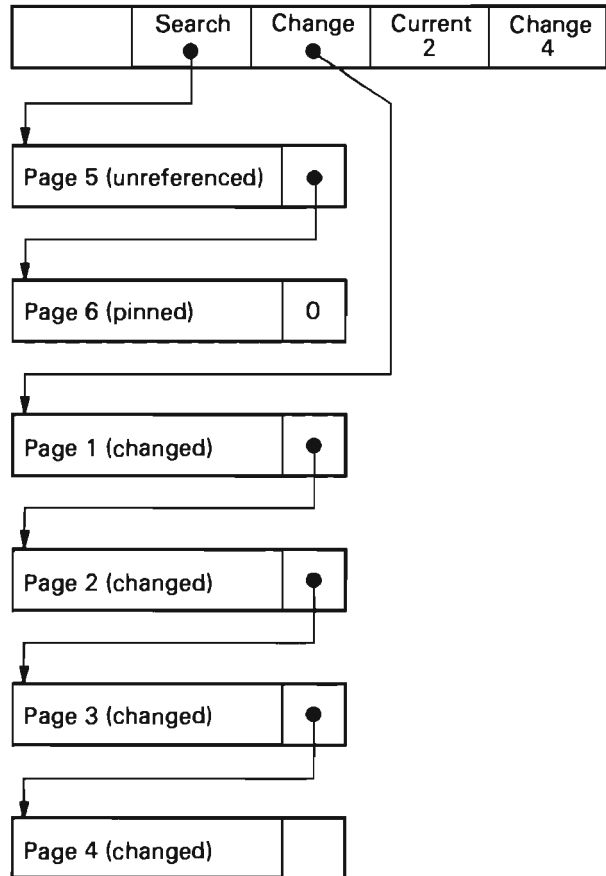


Figure 8-4 (Part 1 of 2). Paging Example

Pool After Page Out

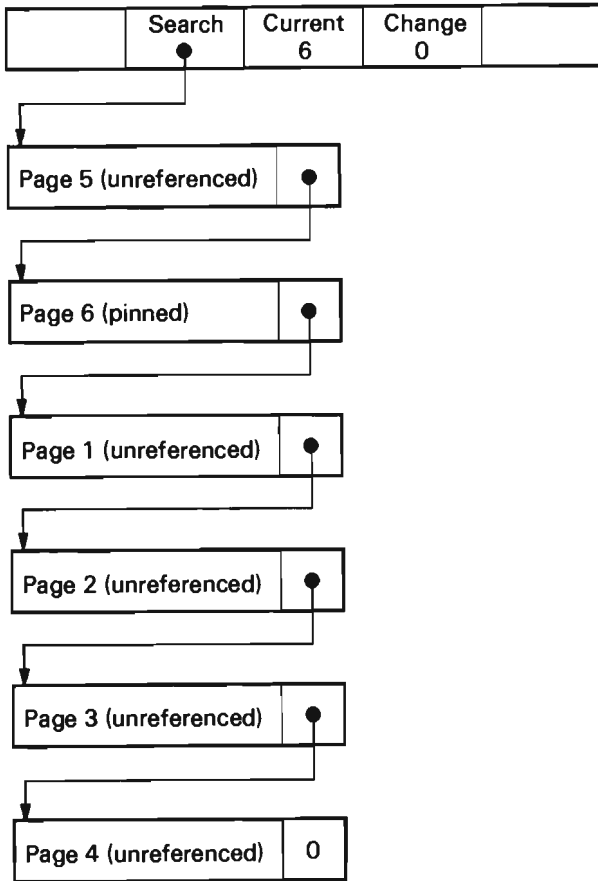


Figure 8-4 (Part 2 of 2). Paging Example

I/O Pending

Because a page can reside only in one place in main storage at any time, it is possible for one request to start operations, followed by another storage management request for the same page before the first request completes. This condition is detected during operational program creation. In this case, the second task is subject to I/O pending processing and proceeds with the common pages according to the following:

Current Request	Earlier Request Indicates I/O Pending
Bring	Overlay
Write, Purge	Quit
Clear	Wait
Remove	Wait

The actions taken are as follows:

- **Overlay:** Read the page even though it is already in main storage, but overlay it with subsequent nonresident pages. Continue processing other pages in the request. If no operational program has been started, however, a wait is done on the pending page.
- **Wait:** The previous I/O operation must complete before processing the pending page.
- **Quit:** Use only pages currently in the operational program for this I/O request. If no operational program has been started, a wait is done on the current page.

The I/O requests that set I/O pending by invalidating the page are as follows:

- **Bring:** Sets I/O pending in the PDE of the pages being brought into main storage by the read.
- **Purge/Write:** Sets I/O pending in the PDE of the pages in the operational program.

Because clear and remove requests do not perform any I/O operations, the MSM directory lock is sufficient to serialize these requests.

When a request is being processed, a wait condition can be detected. This causes the current request to wait for another request to complete. To wait for that request, access pending is set in the PDE of the last virtual address, and the PRE is enqueued to the I/O pending queue in the storage management vector table (SMVT). The pending task then waits on the I/O pending queue.

When I/O complete processing is performed on a request, the access pending bit in the PDE can be set for one or more pages. These bits are reset and the occurrence noted. After the current I/O request is processed, the I/O pending queue is dequeued, member by member. Those members with the last virtual address designated as having I/O pending are returned to the I/O pending queue. If the I/O pending is satisfied, the PRE is sent to the paging request queue of the appropriate task and the task is restarted.

Bring/Purge Access Group

The bring/purge access group functions of Perform Paging Request instruction bring and purge pages of an access group in one operation. They also provide the ability to exclude the reading of pages in the access group that are not heavily used.

The purge access group function checks the access group table of contents. The access group table of contents is brought into main storage and pinned. The shared access group lock is obtained. The disk address of the access group to be purged is found. The first page of the access group to be purged (highest virtual address) is tested for residency and validity. If the page is resident and valid, the reference bit is checked. If the page has not been referenced, history bits are set to 1 in the page entry in the access group table of contents. Setting the history bits to 1 in the access group table of contents prevents the page from being read into main storage during a subsequent bring access group function.

If the information in the page is changed, the page is logically processed. If the information in the page is not changed but is referenced, the page is not written and the history bits in the access group table of contents are set to allow the page to be read on the next bring access group function. This process continues until the entire extent is processed. When the entire extent is processed, the pages are written to auxiliary storage and the processing proceeds with the next extent.

The bring access group function checks each page entry in the access group table of contents for residency. If the page is resident, processing proceeds to the next page. If the page is not resident and the history bits are set to 0, a page frame is allocated and the page is added to the operational program. If the history bits are set to 1, the page is ignored. When all pages in the request have been processed, the pages are read into main storage.

Exchange Bring/Clear

The exchange bring and clear functions are special PPR functions that allow a high degree of control over the use of main storage. A normal bring or clear allocates page frames according to the page replacement algorithm. Exchange bring/clear, however, allows the user of Perform Paging Request instruction to specify page frames, by virtual address, that are to be used as steal candidates.

Under exchange, the second operand of the Perform Paging Request instruction specifies the virtual address of the page(s) to be stolen. Byte register 14 contains the number of pages to be brought/cleared. The second operand address is tested, starting from high to low address, to see if it can be stolen. The page is either cleared (for clear) or added to the operational program (for bring). The virtual address in the range of the first operand is assigned to the stolen frame. If the page in the second operand range is not available, the page replacement algorithm is used to allocate one.

The exchange function allows users the ability to reuse page frames, leaving a larger number of frames available in the system at a given time.

MSM Locks

The following locks are used for main storage management operations. These locks are send/receive counters located in the SMVT. All locks, except as noted, are exclusive.

- **Access Group Directory Lock (exclusive and shared):** This lock synchronizes ASM and MSM operations that involve access groups. When ASM updates either the access group member directory of an access group table of contents, the lock is held exclusively. When MSM interrogates either the member directory or table of contents, MSM holds the lock shared.
- **Truncate Lock (exclusive and shared):** This lock synchronizes ASM truncate and destroy operations with read and clear requests. The lock is held as shared except for remove requests.
- **Permanent and Temporary Directory Locks:** These locks serialize the pageable permanent and temporary directories. These locks are obtained by #SMACDIR when performing a directory lookup.
- **Error Recovery Procedure Lock:** This lock serializes error recovery processing.
- **MSM Lock:** This lock is held while primary directory and the lookaside directory are updated or examined.

Pointer Tags

The machine maintains a bit for each quad-word of main storage. If the bit is 1, a pointer resides in the quad-word. If the bit is 0, the quad-word does not contain a pointer. Only pointer instructions modify the pointer bit.

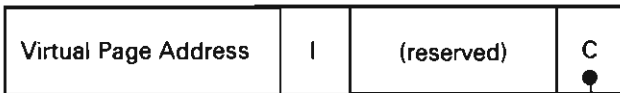
When a page write is performed, the 32 bits representing each of the 32 quad-words associated with that page are also stored. These bits are stored in the pointer. The tags are stored in the structure as shown in Figure 8-5.

The tags are encoded into the structure as follows:

- If the I bit in the storage management header is 0, then the page has no pointers.
- If the I bit is set to 1, then field C (the last 5 bits of the header) represent the location of the first pointer (Format 1 pointer) on the page.

If the left byte in the format 1 pointer is 0, no more pointers are contained in the upper 256 bytes of the page. If the right byte in the format 1 pointer is 0, then no more pointers are contained in the last 256 bytes of the page. If either byte is set to 1's, the corresponding byte offset locates the next pointer in that half page. These pointers (Format 2 pointers) contain the storage management bytes (16 bits) representing the tags for that half page.

**Storage Management
Header Byte**



Page

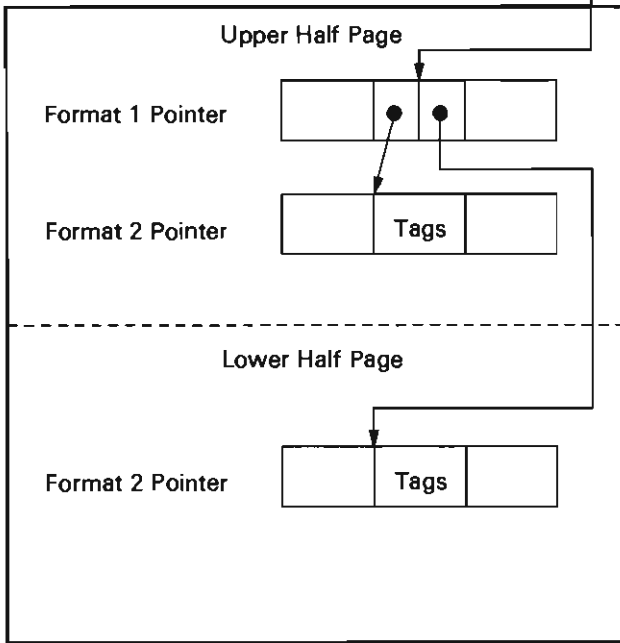


Figure 8-5. Tag Storage Structure

Main Storage Initialization

#SMMSIT in main storage management is the first VMC routine to receive control during the initial microprogram load (IMPL) sequence. This function is loaded as part of the VMC nucleus and performs the following functions:

- Marks PDEs corresponding to bad pages as unusable. Bad pages are indicated in the bad page bit map, built by diagnostics during IMPL.
- Obtains the SMVT information stored on auxiliary storage.
- Determines if directory recovery is required. If required, sets an indicator in the machine initialization status record (MISR). (Directory recovery is actually performed later in the initialization process; see the Initialization portion of the *Machine Support Function* section in this manual for additional information.)
- Empties the temporary directory.
- Initializes PDEs so they are all chained off storage pool 1 (#SMPOOL1).
- Frees the space occupied by the storage management initialization routines (#SMMSIT and #SMPOOL1).

At the completion of this function, certain prebuilt segments can be paged. In particular, VMC, system control adapter (SCA), machine configuration record (MCR), and storage management directory pages can be paged in. If the directory is unusable, it must be rebuilt before other permanent pageable segments can be referenced. After SCA initialization is complete, #SMPICL2 is invoked to verify that drives 2 through 6 (if attached) are operational. Directory recovery can then be performed if required, after which a fully functional paging environment is established. ASM functions (create, destroy segment, and so on) become operational at this time.

DATA AREAS

Access Group

An access group is an object that collects objects into a group that can be operated on by storage management as a unit to reduce disk accesses. It is created as an object with a segment identifier and is allocated a block of contiguous space on storage. Other objects can be allocated within this block, each object having its own virtual address, allowing each object to be accessed individually. However, special directory information enables storage management to transfer all objects within the access group to and from main storage as a single unit.

Permanent Directory

The permanent directory is a machine index that contains 11-, 18-, 21-, and 28-byte entries called ASDEs. The ASDEs map the disk addresses assigned to all permanent segments. ASDEs consist of the first virtual address mapped, followed by one through four extent descriptors. The permanent directory is initialized when VMC is installed and is updated by ASM operations.

Temporary Directory

The format of the temporary directory is the same as the permanent directory except that the temporary directory is for temporary segments.

Access Group Member Directory

This directory maps the member segments in an access group to the owning access group. This enables MSM during a page fault to access objects in an access group when that access group is not in main storage.

Access Group Table of Contents

The access group table of contents describes the contents of an access group in a manner that enables MSM to operate on either the entire access group or its individual pages. The table consists of 8-byte entries; each entry contains the virtual address of a page of an object contained in the access group. Auxiliary storage information and special indicators are also contained in the table of contents.

Lookaside Directory

The lookaside directory is a resident array that contains 11-byte entries (ASDEs with one extent descriptor). These entries are initialized when the permanent or temporary directory is accessed. This directory aids performance by providing a resident least-recently-used type directory of frequently used segments. If a segment is unused for a time and is pushed out of the lookaside directory, the segment is still accessible through either the permanent or temporary directories. The lookaside directory is accessed by an algorithm on the segment identifier. The lookaside directory is updated during MSM operations, and during truncate and destroy ASM operations.

Static Directory

The static directory is a resident table of entries. The entries describe segments that must be accessible without an access to the permanent or temporary directory. (These segments contain the free space, permanent, temporary, and access group directories themselves. The static directory entries also describe VMC code segments.) Static directory entries are identical to lookaside directory entries. The static directory resides in the SMVT and is initialized when VMC is installed. All segments referenced before the storage management directories are recovered must be referenced in the static directory.

Primary Directory

The primary directory describes the current contents of main storage, and is maintained by MSM. The entries in the primary directory are used by hardware during address translation. Entries in this directory are called primary directory entries (PDEs) and contain fields used by HMC, storage management flags, and the auxiliary storage address associated with the virtual storage address.

Storage Management Vector Table

This control block is compiled as a nucleus module (#SMSMVT) and contains the following information:

- Segment identifier generators and segment extender
- Storage management locks
- Device characteristics and free space information
- Work areas for ASM
- Index control blocks for directory operations
- Storage management system-wide statistics
- Main storage pool headers

Portions of the SMVT are preserved on auxiliary storage during shutdown and certain other critical points.

Sector Headers

All sectors on auxiliary storage are 520 bytes long. The first 8 bytes form the header which contains the virtual address of the page and associated page and segment information. The sector header is used to:

- Make pages self-defining so that directories can be recovered.
- Enable MSM to determine whether a page has been previously referenced. If, after a page is transferred to main storage, the header does not match the virtual address being read, MSM assumes that this is the first reference to the page and zero fills the page. In addition to providing 0's in first references, this eliminates the data security requirement of clearing data on destroy operations.

The header also contains information on the location of pointers within the page. The header is read or written with every auxiliary storage operation. For write operations, the information in the header is obtained by MSM from the PDE and from the page itself (tags) so it is not necessary to access the directory to construct the header.

Paging Request Element

The PRE describes the main storage management request. The PRE contains:

- The last virtual address of the request (PRELVADR)
- The length -1 in pages of the request (PRELNGTH)
- The operation request element (ORE) which instructs the paging device (using the operation-unit task) where to read or write page frames to or from
- The PRE request code that describes the operation to be performed
- The task dispatching element (TDE) that will wait for the request

When multiple page I/O operations are to be performed, MSM obtains additional blocks of storage called PRE extensions. The extensions contain a list of the frames to operate on and the header area that the 8-byte record headers are to be either written into or read from. Each extension is 128 bytes long for normal Perform Paging Request instructions and is 512 bytes (one page) for access group requests.

Storage Pools

Storage pools allow the partitioning of main storage page frames. This is done to isolate processes with unlike characteristics from paging against each other, causing degraded response time. In particular, they are intended to prevent batch processes from taking pages from interactive processes. A pool multiprogramming level (MPL) can be used to limit contention for frames within each storage pool.

The storage management portion of a storage pool consists of the following:

- Status
- First frame
- Last frame
- First changed
- Last changed
- Current number
- Maximum number
- Pool statistics

The last 4 bytes of the PDE are used for linking PDEs. A PDE can be on only one storage queue at a time.

Storage Queues (Search and Change)

A storage queue has two components, a head pointer, and a tail pointer. Each storage pool has two storage queues, a search queue that is used to search for the next available frame, and a changed queue that is the page-write-list for that pool.

The change queue is the list of changed pages found by the page replacement algorithm. The page out task treats each of these queue as one large list except that the pool-by-pool list allows the frames to be returned to the same pool after the write is completed.

Two counts are kept:

- The maximum frame count, the count of all frames assigned to the storage pool
- The current frame count, the number of frames currently on the search queue

STRUCTURE

The following is a list of the modules in main storage management and the function that each module performs. The list also shows how the module is invoked.

#CFPOOLS Extract Storage Pool Size

Function: Determines the current size of the process storage pool.

How Invoked: Within this component.

#SMACDIR Access Directory

Function: Determines auxiliary storage address for a page-in request. Sets an exception return code if an invalid address.

How Invoked: Within this component.

#SMCONS Construct Operational Program

Function: Constructs large operational programs for directory recovery and possibly other operational programs for auxiliary storage scan operations.

How Invoked: Within this component.

#SMDALCP Deallocate Page Frame

Function: Returns a frame that was either previously allocated in a call to **#SMALCPF** or one that is part of the resident nucleus and is no longer needed.

How Invoked: User call.

#SMDRTSK Directory Recovery Read Drives Task

Function: Reads auxiliary storage and builds a sequential index of the permanent extent candidates. Immediately recoverable free space is returned to the free space directory. If known defects occur, the associated header is marked as free space and recovery continues.

How Invoked: Within this component.

#SMDR1 Directory Recovery Program, Pass 1

Function: Starts 1 to 4 tasks to read auxiliary storage and synchronizes their finish.

How Invoked: VMC initialization.

#SMDR2 Directory Recovery Program, Pass 2

Function: Rebuilds the permanent directory from the candidate extents found during pass 1. Makes free space entries for failing candidates.

How Invoked: Within this component.

#SMERP Error Recovery

Function: Retries the failing 62PC operation and logs the failure.

How Invoked: Within this component.

#SMFOBTO Function Operation Block (FOB) Time-Out/Halt-Device Processor

Function: Determines when a device times out and then attempts to recover by sending the appropriate channel command to halt the device. When the ORE/FOB with the command is returned, the status is analyzed and error recovery procedures initiated if necessary.

How Invoked: Periodically from the system interval timer.

#SMFRCSG Force Segment Group Utility

Function: Performs a Perform Paging Request instruction write for all changed pages in a specified segment group.

How Invoked: Other VMC components.

#SMMATAS Materialize Auxiliary Storage Data

Function: Obtains auxiliary storage data (from **#SMSMVT**) and stores it in RMMTMD area.

How Invoked: Other VMC components.

#SMMATMS Materialize Main Storage Pool Data

Function: Obtains main storage pool data (from #SMSMVT) and stores it in RMMTMD area.

How Invoked: Other VMC components.

#SMMODAS Modify Auxiliary Storage Controls

Function: Modifies user-specified values for auxiliary storage controls.

How Invoked: Other VMC components.

#SMMODMS Modify Main Storage Pool Controls

Function: Modifies user-specified values for main storage pool controls.

How Invoked: Other VMC components.

#SMMSIT Storage Management Initialization for IMPL

Function: Establishes a paging environment by verifying that the files are operational, the directories are usable, and that main storage is usable.

How Invoked: Initialization.

#SMNERP Error Recovery

Function: Retries the failing 3370 operation and logs the failure.

How Invoked: Within this component.

#SMNERPV Error Recovery Setup

Function: Provides environment independence for module #SMNERP.

How Invoked: Within this component.

#SMPICL2 Auxiliary Storage Startup for Adapter 2 and Drive Check for Drives 2 through 6

Function: Starts the second adapter and checks all file units to determine that they are operational and ready for use by storage management.

How Invoked: VMC initialization.

#SMPOOLI Storage Pool Initialization

Function: Prepares the PDEs for paging.

How Invoked: Within this component.

#SMPOT Page Out Task

Function: Performs the asynchronous page out of changed pages and restarts tasks on the pool wait queue.

How Invoked: This function is an infinite loop and an independent task.

#SMPRE Fixed Allocation PREs

Function: Provides PREs for the various needs.

How Invoked: Not applicable.

#SMSCADP SCA Directory Page

Function: Contains the directory information used by the SCA at initial MPL to locate HMC, diagnostics, and so on.

How Invoked: Not applicable.

#SMSHTDN Storage Management Shutdown

Function: Terminates storage management and preserves the directories. No further auxiliary storage management operations are possible after shutdown, though main storage management functions are still available.

How Invoked: Terminate machine processing or an equivalent call from a disaster cleanup module.

#SMSMVT Storage Management Vector Table

Function: Contains an initialized version of the SMVT and the lookaside directory.

How Invoked: Not applicable.

#SMSMVTI Fix SMVT for Link/Loader

Function: Copies the required values into the SMVT for the link/loader.

How Invoked: Link/loader.

#SMSUSOB Suspend Object Processor

Function: Frees the auxiliary storage by truncating objects to the minimum size that allows ownership and addressability to be retained in the system. After execution of the instruction, the attributes of the target object can be materialized, but any attempt to reference the functional portion results in an exception. The object can be restored by executing the load object function.

How Invoked: Other VMC components.

#SMTAGSE Encode/Decode Pointer Tags

Function: Encodes and decodes pointer tags on a page.

How Invoked: Within this component.

#SVE8PPR Perform Paging Request

Function: Performs the PPR or page fault functions, reads and writes sectors, and allocates virtual-equals-real page frames.

How Invoked: Other VMC components.

Machine Index Management

INTRODUCTION

Indexes provide a means for storing and retrieving data by either content or relative order. Machine indexes are used within the following:

- Independent index management
- Data space indexes
- Context management
- Event management
- Authorization management
- Storage management
- Program management
- Link loader

Machine index operations are invoked as a result of a supervisor linkage. An index control block (IXCB) defines the operation to be performed, the argument to be used, the space for the result, and the location of the index. The calling component must declare and initialize the IXCB prior to invoking machine index management.

Index Structure

Two versions of index code exist. The release 1 version uses 2-byte elements and 512 byte pages and is limited to a single segment group (16 MB). The release 2 version uses 3-byte elements, can contain up to 64 segment groups (1 GB), and allows a variety of page sizes.

Internal machine indexes are binary radix trees that consist of one or more pages and contain the following elements:

- Page pointers: Identify the other pages in the index.
- Text elements: Define the length and address of the associated text on the same page. The following are the types of text elements:
 - Common text: Text that is common to two or more entries. When a common text element is detected during an index search and the text matches the argument, the text is moved to the result area and the argument address is advanced. If the text does not match the argument, the search is terminated.
 - Terminal text: Text that cannot be compressed into common text. There is one terminal text element for each entry in the index.
 - Invalid terminator: Invalid terminators are used to indicate that entries have been removed. The search continues with the adjacent element. The address portion of the text element is changed to an invalid location.
- Nodes: Identify the bit to be tested in the current byte of the argument, and contain a pointer to and a description of a group (cluster) of elements. The following are the types of nodes:
 - Root node: The first node in a page.
 - Successor node: Subsequent nodes in a page.
- Clusters: Contain a left and right element and can contain common text.

Figure 9-1 shows an example of the structure of a machine index and the elements in an index.

An index search is accomplished by using the search argument to select a particular path through the tree from the root node of the trunk page to a terminal text element. Assuming that the argument matches some entry in the index, the path is determined by examining the search argument bits that are specified by the nodes along the path; the root node specifies a bit to be tested and points to a cluster. If the specified bit of the argument is in the zero state, it will use the left element of the next cluster; otherwise, it will use the right element. This process is repeated for each node along the path until a terminal text element is selected. Common text in the selected clusters must match the leading bytes of argument; after this has been verified, the argument is advanced a corresponding number of bytes before performing the bit test. The residue of the argument must match the leading bytes of the terminal text.

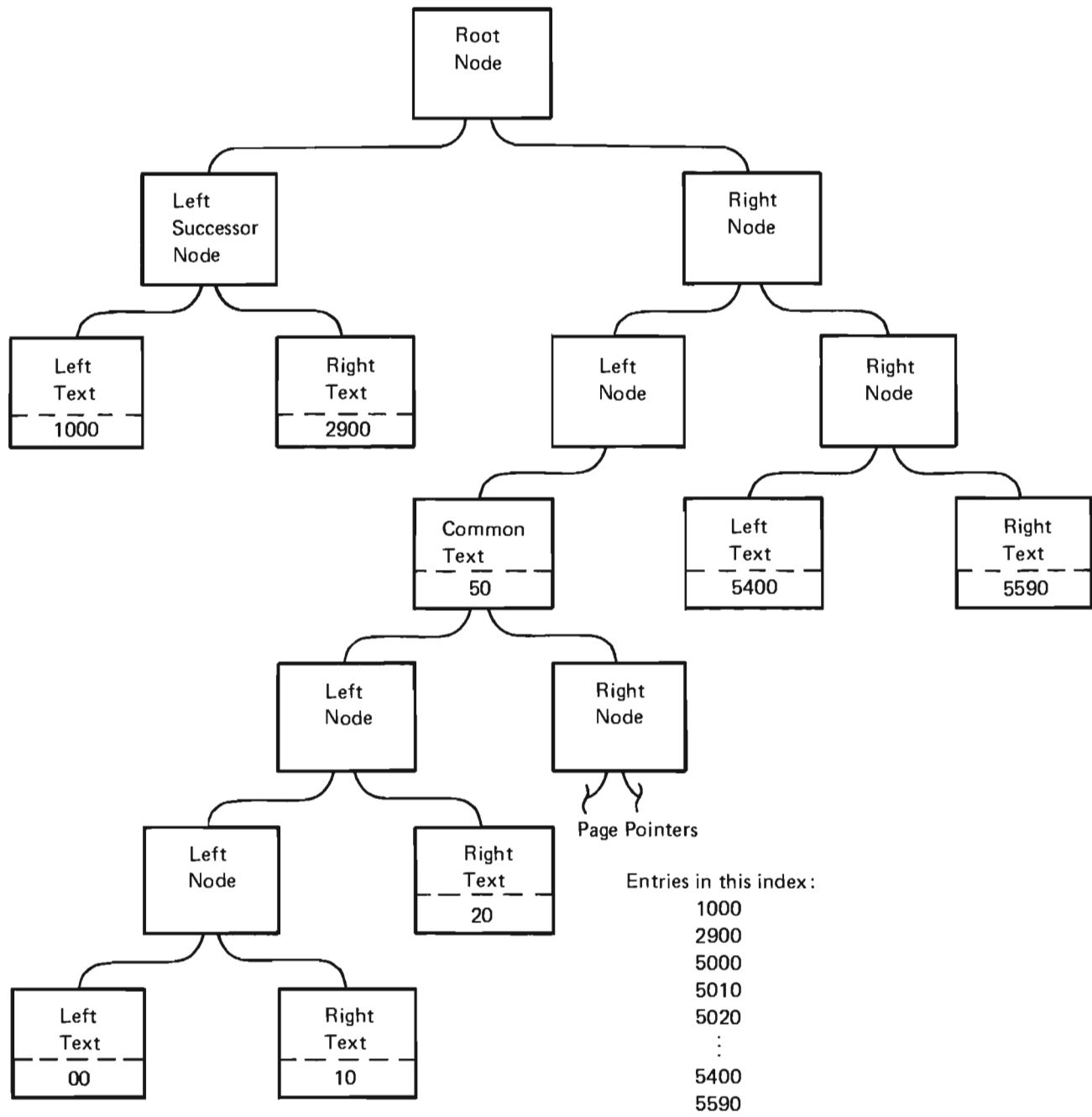


Figure 9-1. Tree Structure

Operations on Machine Indexes

Machine indexes consist of a series of entries. The maximum length of an entry is 128 bytes. The maximum length of an index is 1 GB (64 segment groups). An index entry consists of two parts: a prefix and a suffix.

The prefix is a key used in a find operation to locate an entry. The suffix contains information associated with that key. Following are the basic operations that can be performed on index entries:

- Insert entry
- Find entry
- Remove entry

There are a number of variations that can be specified with each of the preceding operations. These variations are described along with the descriptions of the basic functions.

Find Entry

A find entry operation consists of a generic search of an index to locate an index entry that matches the search argument designated by the IXCB. If a matching entry is found, then either that entry or an adjacent entry (depending on the operation specified in the IXCB) is returned to the area specified for the result. If a matching entry is not found, a flag is set to indicate the mismatch condition.

It is possible for an index to contain multiple entries that satisfy the search requirements. For example, index entries of 5000, 5010, and 5020 all satisfy the requirements if an argument of 50 is specified. If there are multiple entries in an index that satisfy the search requirements, then a specific variation (such as high, low, or next) must be provided to retrieve the desired entry.

Find operations are classed as simple, adjacent, and conditional finds. The classes of find operations and the operation performed are shown in Figure 9-2.

In a simple find operation, the argument is used as a key into the index. Either a matching entry is returned or a mismatch condition is indicated. If multiple entries match the argument, then the default (low or high) entry is selected. Find highest and lowest operations ignore the argument and return either the highest or lowest entry in the index.

In an adjacent find operation, if an entry that matches the argument is found, then either the entry that precedes or follows the matching entry is returned depending on the variation specified (prior or next). If a matching entry is not found, then the entry that would logically precede or follow the argument is returned. The end-of-index condition is returned if no entry in the index could satisfy the requested variation.

In a saved adjacent operation, the argument and result areas are ignored and the entry adjacent to the last found entry is returned in the same result area as the last found entry. The index must not have been modified since the last find. The end-of-index condition is returned if no entry in the index satisfies the requested variation.

Conditional finds perform a simple find on the index and return the entry if a match is found. If a matching entry is not found, then the appropriate adjacent find is performed and the mismatch flag is set in the IXCB.

Class	Operation
Simple	Find low (of equals)
	Find high (of equals)
	Find lowest (in index)
	Find highest (in index)
Adjacent	Find prior
	Find next
	Find generic prior
	Find generic next
Saved Adjacent	Find saved prior
	Find saved next
Conditional	Find low else prior
	Find low else next
	Find high else prior
	Find high else next

Figure 9-2. Find Operations

Figure 9-3 shows an example index and the entry that would be returned for the operation specified using a search argument of 50. The entry 5020 cannot be found using an argument of 50. This entry can be found by using any one of the following operations and search arguments:

Operation	Search Argument
Find low	502 or 5020
Find high	502 or 5020
Find next	5010, 5015, and so on
Find low else prior	502 or 5021
Find prior	503, 5021, and so on

Index Entry	Operation Specified to Return the Entry
1000	Find lowest
4990	Find generic prior
5000	Find low
5010	Find next
5020	
5030	Find prior
5040	Find high
5100	Find generic next
9090	Find highest

Figure 7-3. Example Index Find Operations
(Using an Argument of 50)

Insert Entry

The insert entry operations are as follows:

- Insert entry
- Insert conditionally
- Insert overlay only

The insert entry operation either inserts a new entry into the index or overlays the suffix portion of an existing entry. This operation is performed by first doing a find low operation in order to locate the proper position to insert the entry. If a matching entry already exists in the index, no action is taken (each index entry must be unique). The entry is overlaid with the new entry if the following conditions are met:

- The suffix length is nonzero.
- The mismatch occurred in the suffix.
- There is only one entry in the index with a prefix that matches the search argument.
- The entry has the same length as the argument.

Otherwise, a new entry is added to the index.

For an insert conditionally operation, the entry is inserted into the index if no entries with a prefix that matches the argument are found. If an entry with a prefix that matches the argument is found, then no action is taken except to set the conditional insert flag in the IXCB.

In an insert overlay-only operation, the conditions for an overlay during an insert operation must be met to overlay an entry. No entry is inserted using the insert overlay-only operation if the conditions are not met.

Remove Entry

The remove entry operations are as follows:

- Remove low (of equals)
- Remove high (of equals)
- Remove exact

The first two operations perform a find low or find high operation on the index. In all cases, if a matching entry is located, the entry is deleted. If a matching entry is not located, no operation occurs and a mismatch condition is returned.

DATA AREAS

Index Control Block

Figure 9-4 shows the basic structure of the ICB. The ICB is initialized by the VMC component invoking the internal machine index function. The internal machine index function returns information to the calling VMC component in the ICB at the completion of an index operation. The fields in the ICB and their use are as follows:

- Operation: The index operation to be performed by the internal machine index management function. These operations (and their operation codes in hexadecimal) are as follows:
 - Find low (00)
 - Find high (01)
 - Find prior (02)
 - Find next (03)
 - Find generic prior (04)
 - Find generic next (05)
 - Find lowest (06)
 - Find highest (07)
 - Insert entry (08)
 - Insert conditionally (09)
 - Remove low (0A)
 - Remove high (0B)
 - Find low/else prior (0C)
 - Find low else next (0D)
 - Find high else prior (0E)
 - Find high else next (0F)
 - Insert overlay-only (10)
 - Find saved prior (11)
 - Find saved next (12)
 - Remove exact (13)
- Length of argument area: The length of the area that contains the key to be used in the find operation.
- Length of the suffix area (insert operations only): The length of that portion of the argument area that is to be used as the suffix of the entry.
- Length of result area: The length of the area to receive an index entry after a find operation.
- Length of result returned (output): The length of the entry returned as a result of an index operation.
- Number of pages referenced (output): The count of pages referenced during an index operation after the trunk page.
- Status flags (output): The flags that indicate the result of an index operation. These flags are as follows:
 - Index error encountered
 - Specification error
 - Catastrophic error (damage)
 - Index full
 - Character mismatch
 - End-of-index
 - Insert conditional
- Pages referenced after trunk page (output): Bytes 4 and 5 of the addresses of the first six pages referenced in an index operation. (Two pages for release 2 indexes.)
- Pointer to argument area: A pointer to the area that contains the argument that is to be used as the search key.
- Pointer to the result area: A pointer to the area that is to receive an index entry after a successful index operation.
- Pointer to the trunk page: A pointer to the first page in the index.

Operation ¹
Lengths:
<ul style="list-style-type: none"> • Argument Area¹ • Suffix Area¹ • Result Area¹ • Result Returned²
Number of Pages Referenced ²
Status Flags ²
Pages Referenced after Trunk Page ²
Pointers:
<ul style="list-style-type: none"> • Argument Area¹ • Result Area¹ • Trunk Page¹
Work Area
¹ Input is required for an index operation.
² Information is returned after an index operation.

Figure 9-4. Index Control Block

STRUCTURE

The following is a list of the modules in machine index management and the function that each module performs. The list also shows how the module is invoked.

#IXADPGS Add Pages to a Machine Index

Function: Extends the amount of space available to a machine index.

How Invoked: Other VMC components.

#IXERROR Process Machine Index Error

Function: Creates a VMC log entry if an index error has occurred, and if requested, signals exceptions.

How Invoked: Other VMC components.

#IXEXTDX Extend Index

Function: Extends a release 1 index.

How Invoked: From #IXXEXCB

#IXTRINT Initialize the Trunk Page of a Machine Index

Function: Initializes the trunk page of an index and sets the flag byte to the value contained in the input parameter.

How Invoked: Other VMC components.

#IXXDEST Destroy Secondary Index Segments

Function: Destroys the nonbase segments of a machine index.

How Invoked: Other VMC components.

#IXXEXCB Machine Index Functions

Function: Performs the index operation requested in the index control block (IXCB) for a release 1 index.

How Invoked: Other VMC components.

#IXXEXIX Extend Index

Function: Extends a release 2 index.

How Invoked: From #IXXINDX.

#IXXFIXB Fix Up Base Page

Function: Fixes the base page of a machine index after it has been loaded or moved.

How Invoked: Other VMC components.

#IXXFORC Force Secondary Index Segments

Function: Forces to auxiliary storage the nonbase segments of a machine index.

How Invoked: Other VMC components.

#IXXINDX Machine Index Functions (Extended)

Function: Performs the index operation requested in the index control block for a release 2 index.

How Invoked: Other VMC components.

#IXXWRAP Index Segment Identification (SID)
Wrap Check

Function: Performs SID wrap and other cleanup processing on a machine index.

How Invoked: Other VMC components.

Initialization/Termination Management

INTRODUCTION

Initial Microprogram Load

The initial microprogram load (IMPL) is the method used to load VMC into the machine and begin initializing the machine interface. An IMPL can occur from auxiliary storage or through an external media (alternate IMPL). The operator selects the mode to be used by positioning rotary switches on the console. The IMPL process performs certain hardware diagnostics (some of which can be bypassed through the IMPL abbreviated function) and dispatches an initial task that performs all required VMC initialization functions. These functions include the creation of an initial machine process used to start the first user process.

When the IMPL sequence is initiated by the operator, the horizontal microcode (HMC) performs certain main storage functions (including hardware diagnostics) and dispatches the prime task dispatching queue (TDQ), to which is enqueued a prebuilt task used by VMC to initialize itself and the support functions. This task begins by executing a storage management module (#SMMSIT) as described under *Main Storage Initialization* in the *Main Storage Management* section of this manual. #SMMSIT then branches to the VMC initialization routine #RTVMCIR.

#RTVMCIR first calls the system control adapter (SCA) initialization routine to initialize the system control adapter. The SCA initialization routine starts up the SCA, reads the console rotary switches, and reports initialization conditions to VMC initialization routine (#RTVMCIR) through the machine initialization status record (MISR). The data reported in the MISR includes the position of the console switches, SCA sense and status information, any required error log data, whether or not the primary console and the load/dump device are operational, and if an IMPL halt is to occur. This data can later be accessed by a user through the Materialize Machine Attributes instruction. The temporary and access group directories are then initialized (#SMINDIR). If the directory is unusable and needs to be rebuilt (as indicated in the MISR), the auxiliary storage directory recovery program (#SMDR1) is invoked. When #SMDR1 completes, all segments can be paged. See *Directory Recovery* in the *Auxiliary Storage Management* section of this manual for additional information concerning rebuilding the directory.

The VMC initialization routine checks for the existence of the following objects and structures by referencing them:

- Alter log
- Source/sink active device list
- Data base in-use table
- Object recovery list

If one of the preceding does not currently exist, an exception is signaled, and the IMPL exception handler (#RTIMPLX) is invoked to create the object or structure.

Resource management (**#RMINIT**) initialization is then invoked to create system-wide resources such as task dispatching elements (TDEs), call/return elements (CREs), and machine-wide storage functions. Resource management initialization also starts the machine timer functions and the machine time-of-day clock, using the time saved when the machine was shut down. See *Initialization* in the *Resource Management* section of this manual for additional information concerning resource management.

Exception management initialization (**#EXINIT**) is invoked at this time to initialize statistics recorded about the causes and frequency of effective address overflow exceptions.

Process management initialization routine (**#PMINIT**) sets up the machine-wide index for event management (address is resident in the VMC communications area). **#PMINIT** also initializes two send/receive counters to provide serialization for process management functions and event management indexes. **#PMINIT** initializes the trace table to indicate no active traces. When process management initialization completes, VMC task creation and termination can be performed.

At this point, the VMC log task is created and initialized.

The damage assessment routine (**#RCMKDMG**) is invoked at this time. **#RCMKDMG** runs in conjunction with a special version of the directory rebuild program that reads all pages assigned to permanent segments and marks the permanent directory entries for any segments containing read errors.

#RCMKDMG first checks the storage management vector table (SMVT) to determine if the special version of directory recovery ran previously in the IPL sequence. If not, an entry is made in the VLOG indicating that no further processing was performed, and the routine terminates. Otherwise, the permanent directory is searched for entries marked as containing read errors. If such a segment is found and is part of an object, the object is marked with the appropriate damage, and an entry is made for the object in the object recovery list. A count of the number of damaged objects found is maintained, and that information is entered in the VLOG.

Next, the context rebuild function is called. The machine context and any other context created with the automatic rebuild option are checked for damage. If any of these contexts are damaged, they are rebuilt.

The source/sink initialization function (**#SSINIT**) is then activated. Source/sink initialization builds the machine services control point task. The machine error logging function is activated to log any error messages that exist on the error log queue.

The address regeneration program **#SMSDWRP** is invoked to calculate the number of segment identifiers that have been used. If 95 percent of the segment identifiers have been used, an IMPL halt occurs to notify the operator of this condition. If the operator sets the rotary switches to the appropriate positions and elects to perform another IMPL, **#SMSDWRP** regenerates the segment identifiers to make available all unused segment identifiers, and invokes recovery procedures such as data base recovery. If all segment identifiers have been used when **#SMSDWRP** is invoked, this module performs the identifier regeneration without operator intervention.

The authority management initialization (**#AUINIT**) is then invoked to perform recovery procedures.

The data base, journal, and commit management initialization and recovery routines are invoked to perform recovery procedures and initialization of the data base in-use table. The order of this multiphase process is controlled by **#CFRMAST**.

At any point in the IMPL sequence, errors can occur that can terminate the IMPL sequence. If a certain initialization function cannot complete, it sets indicators to the MISR to indicate to the VMC initialization routine to halt the IMPL sequence.

When a power failure occurs, the machine can be initialized automatically upon restoration of power by the auto-IMPL feature. If the auto-IMPL feature is installed and active, a power failure followed by restoration of power automatically starts the IMPL sequence. The IMPL sequence is the only operation performed. When an auto-IMPL sequence occurs, the storage management directories are automatically rebuilt if required.

Initial Program Load

When the IMPL sequence is complete, the machine is ready to initiate a user process. VMC initiates a temporary machine process called the machine process that is capable of using VMC functions. The machine context is checked during an alternate initial program load (AIPL) and if damaged, it is rebuilt and emptied. If the machine context is usable, a machine user profile is built by #RTUPROF. The address of the user profile is stored in the VMC communications area. Control is then passed from #RTVMCIR to a process management routine #PMIPL1 that builds a TDE for the machine process and places the TDE on the prime TDQ. #RTIPIR gets control as the first program in the machine process and initiates the first user process.

In order to start an initial user process, the following parts are needed:

- A user profile to specify the process authority attributes and objects owned
- A process definition template (PDT) to specify the attributes of the process
- A program template specifying the program to get control in the initial process

If the user profile specified in the template exists, the existing user profile is kept. If the user profile is damaged, it is replaced and addressability to any objects in the profile is lost.

Certain space objects are required:

- A process control space (PCS) for the machine to stack invocations
- A process automatic storage area to stack the invocations
- A process static storage area if the process uses static storage

These areas are further described in *Data Areas* in the *Program Execution Management* section of this manual. The parts can be present on either auxiliary storage or an external media device. If the information is stored on auxiliary storage, an initial program load (IPL) sequence is performed. In this case, a process definition template was previously specified by the use of the Modify Machine Attributes instruction. The process definition template contains system pointers to a user profile and an encapsulated program previously saved within a space object.

If the parts of a process are on the load/dump device, an alternate IPL (AIPL) sequence is performed. The data for an AIPL sequence must be supplied in either load/dump format (a space object) or data interchange format (a character string).

The VMC initial process initiation routine uses the process definition template to initiate a process. If an alternate IPL sequence is to be performed, a load/dump session is established and the data is read from the load/dump device; otherwise the data is located on auxiliary storage via the initial process definition template. The data read from the load/dump device is a series of creation templates for objects. These templates are:

- A user profile template for a create user profile operation.
- A program template.
- The spaces needed for user process initiation. (These spaces are created with default attributes by the initial process initiation routine.)
- A process definition template.

The spaces are created with the Create Space instruction. The system pointers are stored in the machine initialization status record. The program is translated using the Create Program instruction, and a user profile is created using the Create User Profile instruction. The user program and the user profile are then used by the Initiate Process instruction to create the first user process. The machine process then signals the VMC initialization process, and the machine process destroys itself.

If the IPL or AIPL sequence cannot be completed because an error is detected, an IPL halt is executed and, with the exception of a damaged user profile error, no effort is made to restart the IPL sequence. The IPL halt performs the following:

- Sets indicators and reason code in the MISR and writes the MISR to auxiliary storage
- Destroys any objects created by an IPL
- Executes the storage management shutdown function
- Places the machine in checkstop state and places a termination code in the sequence indicator lights

An IPL halt can be caused by a hardware error (a machine check or initial microprogram load (IMPL) exception) or an exception signaled by a function used by the IPL sequence.

Figure 10-1 shows the order of activation of VMC components that initialize the VMC support functions.

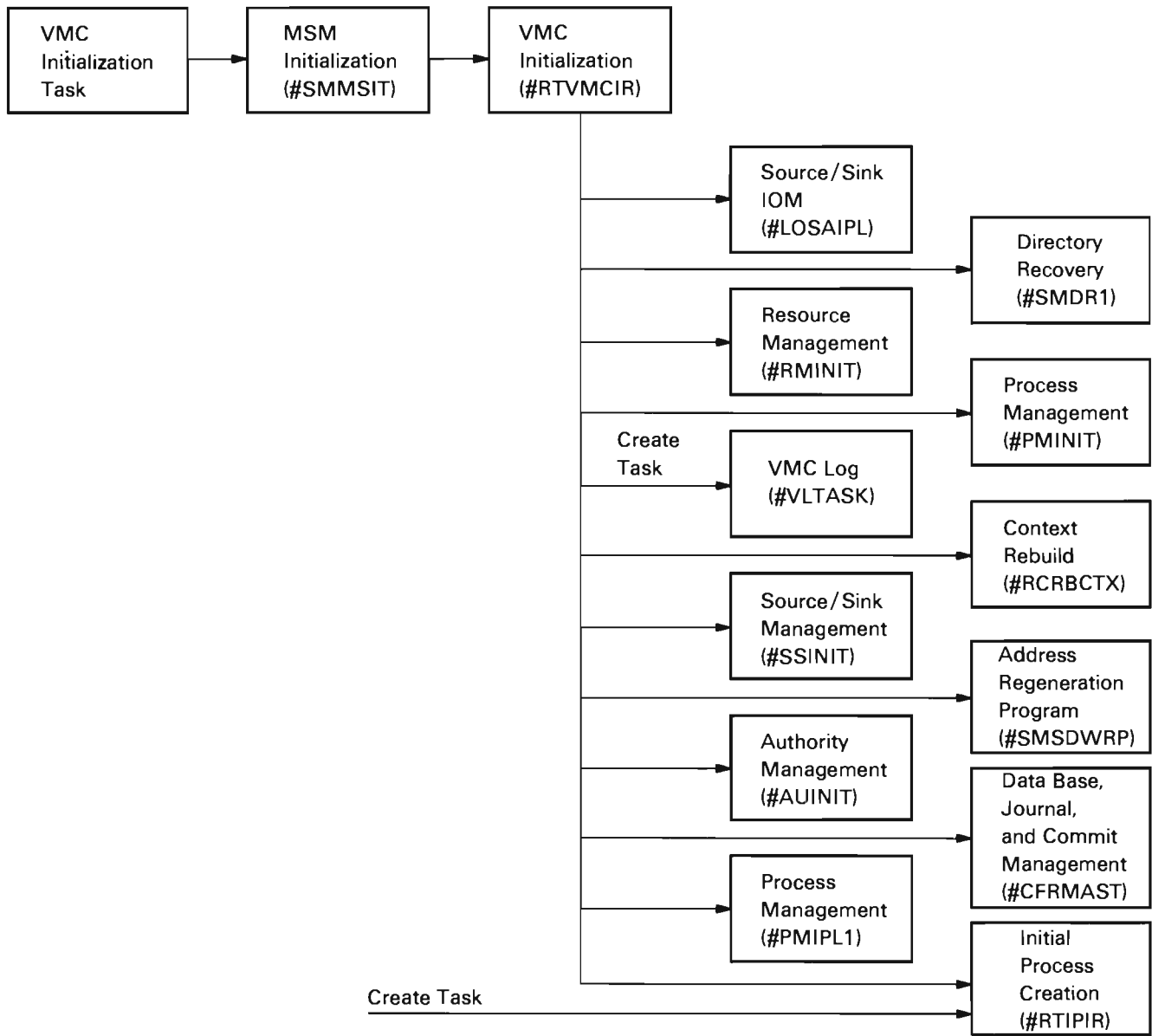


Figure 10-1. Initialization Sequence

Terminate Machine Processing

The terminate machine processing function provides the capability of destroying all processes in the system and either turning off the power to the machine or putting the machine into the check stop state. The processes are destroyed in a manner as defined by the Terminate Process instruction. The same function as defined by the Terminate Process instruction is done for each individual process that is destroyed. This operation is further described in the *Process Management* section of this manual.

The process that invokes the terminate machine processing function must have the authority to destroy all processes within the machine. The terminate function is invoked whenever the Terminate Machine Processing instruction is issued. A fixed time interval elapses before machine processing is terminated. This allows all processes to complete to normal termination. If one or more processes remain active at the end of the time interval, those processes are abnormally terminated. Machine processing then completes processing using a separate internal task.

Once machine processing has been terminated, it can only be reactivated through machine initialization.

An optional function performed by the terminate machine processing function is that of turning off the machine power supply. 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 terminate functions is to put the machine into the check stopped state. For diagnostic purposes, the process that invoked the terminate function can save a space pointer for a permanent space object that contains diagnostic data. This data is user-defined and is not used by the machine.

DATA AREAS

VMC Communications Area (YYVCA)

The VMC communications area contains the various pageable control areas used in VMC. An overview of the VMC communications area is shown in Figure 10-2. The VMC communications area contains pointers used by VMC components to address structures that are pageable.

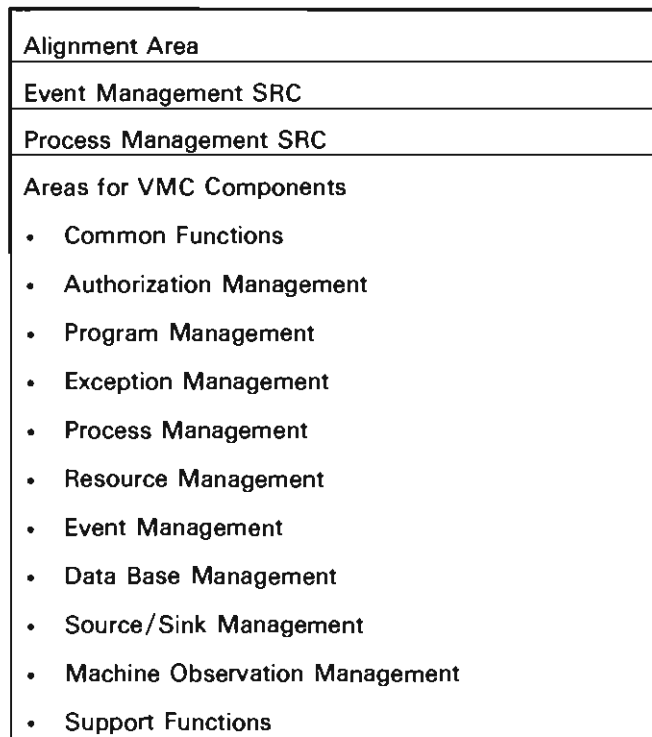


Figure 10-2. VMC Communications Area

Machine Initialization Status Record (YYMISR)

The machine initialization status record (MISR) is used by VMC components to store information relating to the status of the machine following an IPL or IMPL. Some of the contents of the MISR can be materialized by using a Materialize Machine Attributes instruction. An overview of the MISR is shown in Figure 10-3.

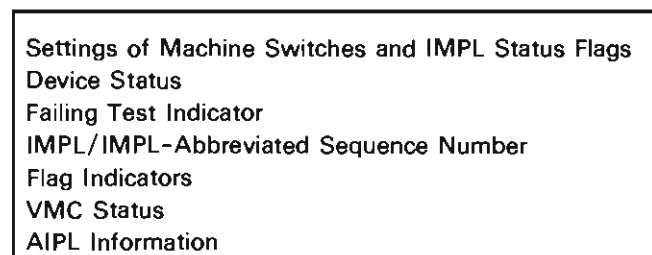


Figure 10-3. Machine Initialization Status Record

Object Recovery List

The object recovery list identifies the objects that were not completely processed at machine termination. A recovery entry exists in the list for every object subject to recovery activity. These entries contain the object type, a pointer to the object, and the status of the object.

STRUCTURE

The following is a list of the modules in machine initialization management and the function that each module performs. The list also shows how the module is invoked.

#RTIAIPL Read AIPL Data

Function: Reads AIPL process creation data from the load/dump device.

How Invoked: Within this component.

#RTIMPLX IMPL Exception Handler

Function: Rebuilds damaged or missing segments.

How Invoked: Within this component (exceptions generated in #RTVMCIR).

#RTIPIR Initial Process Initialization Routine

Function: This routine is activated as a machine program that executes in a process environment. This routine creates a user defined process from data either on the load/dump device or in the machine.

How Invoked: Process Management.

#RTIPLHT IPL Halt

Function: Cleans up the IPL sequence on an abnormal termination.

How Invoked: Exception management when an error occurs during the IPL or AIPL.

#RTMCR Machine Configuration Record

Function: Contains the initialized machine configuration record. This module does not contain executable code.

How Invoked: Not applicable.

#RTMPT1 Terminate Machine Processing Part 1

Function: Starts the machine processing termination function.

How Invoked: Other VMC components.

#RTMPT2 Terminate Machine Processing Part 2

Function: Destroys all user processes and stops the machine.

How Invoked: Within this component.

#RTMPT3 Terminate Machine Processing Part 3

Function: Invokes the destroy function for a process.

How Invoked: Other VMC components.

#RTVCA VMC Communications Area Generating Data

Function: Contains the data used to generate the VMC communications area. This module does not contain executable code.

How Invoked: Not applicable.

#RTVMCIR VMC Initialization Routine

Function: Performs VMC initialization at IMPL or IMPL-abbreviated time.

How Invoked: Other VMC components.



Machine Check Management

INTRODUCTION

Machine check management protects the System/38 instruction set user from error conditions caused by hardware, microcode, and logic failures. When a machine check occurs, machine check management performs one of the following:

- Immediately halts machine processing when a failure is severe enough (terminal) that processing cannot continue
- Initiates shutdown
- Generates either a machine check event and exception or a function check exception

Machine check management also records certain information for diagnostic purposes. An error report is sent to the machine error log or the VMC log for all function and machine checks except immediate machine checks.

Machine checks occur as follows:

- As a result of a hardware failure in the processor, a microcode failure caused by misuse of the internal microprogramming (IMP) instructions, or an exceptional condition in the execution of a microcode routine. These are hardware-reported machine checks that are stored in the machine error log.
- As a result of an unexpected IMP exception in a VMC routine or a condition detected by a VMC-reported function and machine checks that are stored in the VMC log.

Hardware machine checks are reported by the hardware machine check handler which is a horizontal microcode (HMC) component. When a machine check is to be reported, the hardware machine check handler disables the task dispatcher and fills the machine check logout buffer with error information and task status (base registers, instruction address register, condition codes, instruction length code). Machine check mode is set, which forces a hard machine stop if another machine check or exception occurs before machine check mode is reset. The hardware machine check handler then passes control to the resident VMC machine check handler to determine the severity of the error and to determine if further processing can take place.

Normally, when the resident machine check handler is invoked, it runs under the currently executing task dispatching element (TDE). The resident machine check handler can also be invoked while in a wait state with no TDE executing. In this case, the resident machine check handler enters the run state, copies the error information from the machine check logout buffer to a machine check queue element, and returns to the wait state. The error is recorded when another machine check occurs.

The resident machine check handler determines if the machine check is hard or soft as follows:

- **Hard Machine Check:** A machine check that requires some recovery action by VMC before processing can continue. (For some hard machine checks, recovery may not be possible.)
- **Soft Machine Check:** A machine check that requires no recovery action by VMC (a hardware function may have already performed recovery actions).

If the machine check is hard, the following conditions cause an immediate hard stop:

- The machine check occurred while task dispatching was disabled
- An error in a horizontal microcode routine
- Certain channel errors
- Certain main storage locks active

These conditions prevent further machine processing because main storage management will not function. If none of these conditions occur, the resident machine check handler prepares to invoke the pageable machine check handler. A call/return element (CRE) is obtained to save the task status, and the remainder of the machine check logout buffer is recorded in a machine check queue element. An error in any of these operations causes a hard stop. When processing is complete, the dispatcher is enabled, machine check mode is reset, and the resident machine check handler invokes the pageable machine check handler. (Because the majority of machine check management code is not in main storage, a portion of storage management must be usable before machine check management can proceed.)

The pageable machine check handler establishes a normal VMC execution environment and attempts to recover the error. If an invocation work area (IWA) is not present, one is created. If a main storage failure occurred, the operation is retried by attempting to load from the failing location. If no additional machine check occurs, the retry of the failing location is considered successful. If the retry of the failing locations is successful or if no main storage error occurred, the pageable machine check handler determines if the failing instruction can be retried. Only those instructions that can be executed a number of times with identical results can be retried. If the retry is successful (no machine check results), the task is restarted (via the Supervisor Exit instruction) at the point of the failure.

If the retry did not succeed (because of a failing storage location, nonretrievable instruction, or instruction retry failure), the following occurs:

- The error is logged to the error log.
- A machine check event is signaled and an entry is made in the machine error file (regardless of the disposition of the error).
- A machine check exception is signaled to the machine interface user through the normal exception handling process.
- The task will probably be terminated, unless VMC invocations handle or exception handlers process the error.


If a soft machine check is reported to the resident machine check handler, normal machine check processing without retry occurs. The error is reported and the task is restarted.

The following actions are taken in machine check processing to maintain the integrity of main storage:

- When a hard main storage failure occurs, the primary directory entry corresponding to the bad frame is marked as bad. The frame can still be accessed, but it will not be stolen or relocated by main storage management.
- When attempts to recover a main storage error fail, the frame is invalidated and addressability to the frame is destroyed.
- The location of a bad frame is tested. If the frame is permanently resident or pinned, a hard stop is executed.
- If a bad frame has been changed (that is, an exact copy does not exist on auxiliary storage), an attempt is made to signal a machine check and to invalidate the corresponding page on auxiliary storage. If this attempt fails, machine processing is terminated.

If none of these conditions has occurred, the page is recovered by relocating the data to another page frame.

VMC checks caused by program execution errors are reported as function or machine checks through the exception management or the machine check handler. If an IMP exception occurs in a VMC program and the exception is not handled by a VMC exception handler, the exception is converted to a hard machine check by the exception handler and the appropriate recording occurs. The exception handler then reports a machine check exception to the user. A VMC routine can also explicitly create a machine or a function check by signaling a machine check.




Throughout machine check processing, the integrity of the machine interface and protection of data is the primary concern. If the machine check handler cannot isolate the error, machine processing is terminated. In severe cases when the machine check handler itself cannot operate, the cause of the error can usually be determined by looking at the machine check logout buffer in main storage. If the resident machine check handler is forced to execute a hard stop, the logout buffer is preserved.

DATA AREAS

Machine Check Logout Buffer (RTMCLB1)

The machine check logout buffer is an area in main storage that is reserved for the reporting of machine checks by the hardware machine check handler. This area contains data concerning the cause of the machine check and the environment existing when the machine check occurred.

Machine Check Queue



The machine check queue is a queue in main storage used to transfer machine check queue elements between the resident and pageable machine check handler.

Machine Check Queue Element (RTMCQE1)

Machine check queue elements are used by machine check to pass machine check related information. An element contains the following information:

- Element description
- Hardware error statistics
- HMC logout data

STRUCTURE

The following is a list of the modules in machine check management and the function that each module performs. The list also shows how the module is invoked.

#RTELOG Send Machine Check to Error Log

Function: Puts header on machine check error message and sends the message to the error log.

How Invoked: Within this component.

#RTIBADP Resident Bad Page Recovery

Function: Attempts to retry certain bad main storage frames accessed by main storage management.

How Invoked: Within this component.

#RTIMCH Resident Machine Check Handler

Function: Processes machine malfunctions detected by the HMC.

How Invoked: From HMC.

#RTIRTRY Resident Bad Page Recovery Recording Routine

Function: Records bad frames recovered by #RTIBADP.

How Invoked: Resource management service task.

#RTPMCH Pageable Machine Check Handler

Function: Retries hardware errors, logs errors, initiates dump, and invokes the third-level exception handler (TLEH).

How Invoked: Within this component.

#RTPMCKH Machine Check Service Routine

Function: Provides service functions for machine check processing.

How Invoked: Within this component and exception management.

#RTPMCKX Machine Check Component-Specific Exception Handler (CSEH)

Function: Processes exceptions that occur in the pageable machine check handler.

How Invoked: Other VMC components.

Machine Observation Management

INTRODUCTION

Machine observation management provide a means for the user to view detailed information relating to system objects and to trace the occurrence of the execution of specific System/38 instructions. These functions are accomplished through the following System/38 instructions:

- Materialize System Object
- Materialize Pointer
- Materialize Pointer Locations
- Trace Instructions
- Cancel Trace Instructions
- Trace Invocations
- Cancel Trace Invocations
- Materialize Invocation
- Materialize Instruction Attributes

Materialize System Object

#DOMTSOB is invoked as a result of a Materialize System Object instruction. This module invokes #CFOCHKR to validate the operands, authorizations, and lock enforcement. #DOMTSOB then materializes information about the system object addressed by operand 2. This information includes context and user profile information for objects that are addressed by contexts and user profiles.

Materialize Pointer

#DOMTPTR is invoked as a result of a Materialize Pointer instruction. This module first validates the instruction operands. #DOMTPTR then materializes information about the pointer contained in operand 2. This information includes context information for pointers to objects that are addressed by a context.

Materialize Pointer Locations

#DOMTPTL is invoked as a result of a Materialize Pointer Locations instruction. This module validates the operands. #DOMTPTL then materializes information about the space addressed by operand 2. This information is presented using the following algorithm:

- Each bit in the area addressed by operand 1 represents 16 bytes of data in the area addressed by operand 2.
- A bit value of 0 in the materialize area (operand 1) indicates that the corresponding 16 bytes of the scanned area (operand 2) did not contain a pointer.
- A bit value of 1 in the materialize area indicates that the corresponding 16 bytes of the search area contains a pointer.

Trace Instructions

#DOTRINS is invoked as a result of a Trace Instructions System/38 instruction. This module first validates the input operands. If the trace table does not currently exist, it is created. If the trace table does not exist, it is necessary to check that the VMC service function is not using the program event monitor (PEM). If the PEM is in use, then an exception is signaled; otherwise, the PEM is marked as being in use by the trace instructions function. #DOTRINS then creates a program trace element for the specified program. The trace element contains the low and high PEM ranges for that program. #DOTRINS then determines the hardware address for each instruction number and creates a trace point entry. The low and high PEM limits for the program are then set.

#DOTRSEV is invoked from the exception handler (#SV00EXC) when a PEM exception is signaled. #DOTRSEV searches the trace for a machine hardware address. If a machine address is found, an instruction reference event is signaled and a Supervisor Link Monitored instruction is executed. The Supervisor Link Monitored instruction allows the event monitor to receive control. #DOTSEV then enables the PEM and allows the monitored instruction to be executed.

#DOTRINX is invoked if a call or return is executed when instruction tracing is active. #DOTRINX determines what program is about to be invoked if any trace points have been specified in the trace table for that program. If trace points have been specified, #DOTRINX sets a flag in the task dispatching element (TDE) to indicate that a PEM is active, and sets the low and high PEM range for this program. When a hardware instruction within the specified ranges is executed, a PEM exception is signaled.

Cancel Trace

#DOCTRIN is invoked as a result of a Cancel Trace Instructions System/38 instruction. This module validates the operands in the instruction. #DOCTRIN then determines the hardware address for each System/38 instruction specified in operand 2. #DOCTRIN scans the trace table and deletes the entries that match the instruction list, and sets the high and low PEM limits for the specified program.

Trace and Cancel Trace Invocations

#DOTRCLE contains entry points to trace invocations and to cancel the trace, and to signal the invocation-reference event. #DOTRINV is invoked as a result of a Trace Invocations instruction. This module sets invocation trace flags in the specified invocation control block. #DOCTRNV is invoked as a result of a Cancel Invocations Trace instruction. This module resets invocation trace flags in the specified invocation control block.

#DOTRIEV and #DOTRCEX are invoked during call and return functions. These modules build event related data and signal an invocation-reference event.

Materialize Invocation

#DOMATIA is invoked as a result of a Materialize Invocation instruction. This module first validates the operands of the instruction. The module then returns information about the specified invocation and, optionally, returns argument addresses and exception descriptions.

Materialize Instruction Attributes

#DOMTINS is invoked as a result of a Materialize Instruction Attributes instruction. The module first checks the input template for obvious validity problems. Then the program is scanned until the desired instruction is located. Finally, each operand of the desired instruction is materialized.

DATA AREAS

Trace Table

The trace table contains information required to trace the instructions in a process. This table retains the information about user programs. The trace table is located by a pointer in the process control block for a process. The table contains the following information:

- A program trace element for each program being traced. (The element contains the high and low PEM limits.)
- A list of all System/38 instructions that are being traced and the corresponding hardware address for those instructions.

If there are too many instructions to be retained in the trace table, a machine index is created to retain the trace points for some programs.

STRUCTURE

The following is a list of the modules in machine observation management and the function that each module performs. The list also shows how the module is invoked.

#DOCTRIN Cancel Trace Instructions

Function: Removes the specified instructions from the trace table.

How Invoked: As a result of a Cancel Trace Instructions instruction and other VMC components.

#DOMATIA Materialize Invocation

Function: Returns the program and instruction numbers of the specified invocation, and returns the argument addresses and exception descriptions.

How Invoked: Materialize Invocation instruction.

#DOMODSO Modify System Object

Function: Modify the time-stamp in the EPA header to the current value.

How Invoked: SVL router.

#DOMTINS Materialize Instruction Attributes

Function: Materializes the attributes of an instruction for a specific invocation within the process issuing the instruction.

How Invoked: Materialize Instruction Attributes instruction.

#DOMTPTL Materialize Pointer Locations

Function: Materializes the symbolic locations of valid pointers in a given string of data.

How Invoked: Materialize Pointer Location instruction.

#DOMTPTR Materialize Pointer

Function: Materializes the type and attributes of a pointer.

How Invoked: Materialize Pointer instruction.

#DOMTSOB Materialize System Objects

Function: Materializes the identity and size of the system object addressed by the system pointer.

How Invoked: Materialize System Objects instruction.

#DOTRCLE Trace Invocations

Function: Modifies trace invocation flags and signals the trace-invocations event.

How Invoked: Trace Invocations and Cancel Invocations Trace instructions and other VMC components.

#DOTRINS Trace Instructions

Function: Initiate trace instructions for the System/38 instructions.

How Invoked: Trace Instructions instruction.

#DOTRINX Set PEM Range

Function: Determines the correct PEM range for a call or return.

How Invoked: Other VMC components.

#DOTRSEV Trace Instructions Signal Event

Function: Signals trace-instructions events for those event-monitored exceptions that occur at addresses specified in the trace table.

How Invoked: Other VMC components.



Service and Installation Management

INTRODUCTION

VMC provides tools for use by service personnel for problem determination. The service functions provided are as follows:

- Virtual Storage Standalone Dump: Dumps selected virtual storage to a diskette.
- Print Standalone Dump: Prints all or selective portions of either main or virtual storage dumps.
- VMC Log: Dumps selected VMC log entries to either a diskette or a printer, and modifies the characteristics of the VMC log.
- Display/Alter/Dump: Displays and alters the contents of virtual storage and dumps virtual storage to a diskette or a printer.
- VMC Trace: Provides a chronological record of the execution of selected events within VMC.
- Address Stop/Instruction Step: Provides address stop and instruction step capabilities.
- Machine Configuration Record Update Facility: Assists the user in updating the machine configuration record (MCR).
- Link/Loader: Copies the microcode from the diskette magazine to the area reserved for this code during auxiliary storage initialization.
- Auxiliary Storage Initialization: Initializes auxiliary storage and builds the storage management directory.

The internal operations of these functions are not described except to provide a list of the modules that perform these functions. For instructions on the use of the service aids, see the *System/38 Diagnostics Aids* manual and the *System/38 Service Guide*.

STRUCTURE

The following is a list of the modules that perform the service and installation functions. The function that each module performs is also included.

#CFMLOG	Performs VMC log operations
#RIADRSM	Addresses stop/instruction step service monitor linkage
#RIADRST	Addresses stop/instruction step service function
#RID52	Diagnoses machine interface dump list
#RIDAADR	Displays/alters/dumps address select
#RIDADAT	Displays/alters/dumps data display/alter support
#RIDADDS	Displays/alters/dumps display dump status
#RIDADSL	Displays/alters/dumps data select
#RIDA EHS	Displays/alters/dumps component-specific exception handler (CSEH)
#RIDAFSL	Displays/alters/dumps function select
#RIDAINT	Displays/alters/dumps initialization
#RIDAIXU	Displays/alters/dumps index utility function
#RIDAMIO	Displays/alters/dumps object select
#RIDAMOD	Displays/alters/dumps VMC module select
#RIDARTE	Displays/alters/dumps dump router
#RIDASM1	Displays/alters/dumps screen and message library
#RIDASM3	Displays/alters/dumps screen and message library
#RIDASM4	Displays/alters/dumps screen and message library

#RIDASM6	Displays/alters/dumps screen and message library	#RIPEMEX	Addresses stop/instruction stop and trace instruction program event monitor exception handler
#RIDASM8	Displays/alters/dumps screen and message library	#RIRICTL	Retrieves internal data
#RIDASUB	Displays/alters/dumps subtask (processes dumps)	#RIRIBUF	Traces retrieve buffer
#RIDATKS	Displays/alters/dumps task/process select	#RITOUCH	Traces segment identifier maintenance
#RIDAVMD	Displays/alters/dumps VMC data	#RITRACE	General trace collection
#RIDDIAGF	Diagnoses instruction router	#RITRCAC	Activates trace
#RIDNDPI	Diagnoses dump process internal	#RITRCAL	Allocates trace space
#RIDNECI	Diagnoses engineering change inquiry	#RITRCCL	Clears tracing
#RIDPRSM	Prints standalone dump main/virtual storage dump screen and message library	#RITRCDA	De-activates trace
#RIDPR2	Prints standalone dump main/virtual storage dump	#RITRCDP	Traces dump control
#RIDPR21	Prints standalone dump main storage dump initialization	#RITRCSC	Traces common scroll
#RIDPR22	Prints standalone dump main storage dump	#RITRCSM	Traces control
#RIDPR23	Prints standalone dump virtual storage dump initialization	#RITRMI	Traces control from machine interface
#RIDPR24	Prints standalone virtual storage dump	#RITRSAC	Traces activation (screen interface)
#RIDPR25	Prints standalone dump main storage dump get page	#RITRSAD	Traces activate/de-activate source/sink object
#RIDPR26	Prints standalone dump main storage dump task dispatching element (TDE)/call/return element (CRE) chains	#RITRSAL	Traces allocation (screen interface)
#RIGTBUF	Gets trace recording buffer	#RITRSCL	Traces clear (screen interface)
#RIMATMA	Materializes machine attributes	#RITRSDA	Traces de-activate (screen interface)
#RIMODMA	Driver for modify machine attributes	#RITRSDP	Traces dump control (screen interface)
#RIPEMCK	Validates data to set program-event monitor	#RITRSM1	Traces screen and message library part 1
		#RITRSM2	Traces screen and message library part 2
		#RITRSM3	Traces screen and message library part 3

#RITRSM4	Traces screen and message library part 4	#SDDIDC	Dumps compress
#RITRSM5	Traces screen and message library part 5	#SDDIDIX	Dumps machine index entry
#RITRSM6	Traces screen and message library part 6	#SDDIDPR	Dumps print
#RITRSM7	Traces screen and message library part 7	#SDDIDSG	Dumps segment
#RITRSM8	Traces screen and message library part 8	#SDDIFS0	Format search table (0)
#RITRSM9	Traces screen and message library part 9	#SDDIFS1	Format search table (1)
#RITRSSC	Traces scroll control (screen interface)	#SDDIFS2	Format search table (2)
#RITRSSI	Traces source/sink object initialization	#SDDIFS3	Format search table (trace)
#RITRSTS	Traces status (screen interface)	#SDDIFS4	Format search table (object specific header)
#RITRSTS	Traces task switch save buffer	#SDDIFS5	Format search table
#RIVLDMP	VMC log asynchronous dump subfunction	#SDDIFS6	Format search table
#RIVLSF	VMC log service function	#SDDIFS7	Format search table
#RIVLSML	VMC log screen and message library	#SDDIFS8	Format search table
#SDAIBLD	Services function driver address index build	#SDDIGDS	Dumps record interchange get
#SDALINQ	Displays/alters/dumps alter log inquiry	#SDDIMOV	Dumps interchange move data
#SDALINS	Displays/alters/dumps log insert	#SDDIPDS	Dumps record interchange put
#SDCDCTL	Common display	#SDDIPF	Dumps interchange print formatted
#SDCNCVT	Converts data	#SDDIPH	Dumps interchange print hex
#SDCNMOV	Moves data with exception handler	#SDDIPL	Dumps interchange put line
#SDCNTIM	Services function driver convert time	#SDDIPOC	Dumps interchange print open/close
#SDDCCTL	Services function driver task controller	#SDDIP21	Program special print routine (for machine interface programs)
#SDDIBLD	Dumps entry string build	#SDDIP22	Context entry special print routine
#SDDICTL	Dumps control	#SDDIP23	User profile entry special print routine
		#SDDIP49	Traces special formal routine
		#SDDISDT	Dumps structure

#SDDISRT	Dumps routing table	#SDDIS34	Dumps cursor
#SDDISTB	Instruction mnemonic tables	#SDDIS41	Dumps machine wide storage
#SDDIS01	Dumps task/process	#SDDIS49	Dumps trace tables
#SDDIS02	Dumps object	#SDDIS50	Dumps task chain
#SDDIS03	Dumps process control block (PCB)/task control block	#SDDIS55	Dumps index control block
#SDDIS04	Dumps default structure	#SDDIS88	Dumps alter log
#SDDIS05	Dumps task chain	#SDDIS90	Dumps process static storage area/process automatic storage area
#SDDIS06	Dumps machine index	#SDDIS91	Dumps MCR
#SDDIS07	Dumps segment	#SDDIS92	Dumps machine check logout buffer
#SDDIS14	Dumps seize/lock	#SDDIS93	Dumps task summary
#SDDIS16	Dumps link map	#SDDIS94	Dumps task dispatching element and call return element
#SDDIS20	Dumps access group	#SDDKCTL	Handles diskette I/O
#SDDIS21	Dumps program mainline	#SDDPCTL	Handles data path
#SDDIS22	Dumps contexts	#SDDSBLD	Screens build
#SDDIS23	Dumps user profile	#SDDCLS	Displays close
#SDDIS24	Dumps user queue	#SDDSCTL	Displays open/close/read/write
#SDDIS25	Dumps data space	#SDDSOPN	Displays open
#SDDIS26	Dumps data space index	#SDDSPFK	Asynchronous program function keys on/off
#SDDIS27	Dumps independent index	#SDFCABR	Requests a service function
#SDDIS28	Dumps logical unit description (LUD)	#SDFCCTL	Services function controller
#SDDIS29	Dumps controller description (CD)	#SDFCDST	Destroys a service function
#SDDIS30	Dumps network description (ND)	#SDFCGI	Handles general inquiry request I/O messages
#SDDIS31	Dumps space object	#SDFCPFk	Handles program function key response
#SDDIS32	Dumps machine context	#SDFCRSP	Processes response request I/O
#SDDIS33	Dumps process control space (PCS)		

#SDFCSTR	Starts a service function	#SDTCERA	Common module to handle errors
#SDFCTMD	Handles service function termination message	#SDTCMOD	Mainline for all service function driver tasks
#SDFCTRM	Requests a service function	#SDTCNDT	Terminates service function driver tasks
#SDFMCTL	Finds machine interface object common function	#SDTCRTT	Services function driver request task termination
#SDFVMOD	Services function driver find VMC module	#SDTKSEL	Task/process selection menu
#SDIFTAB	Services function table	#SDTKSML	Task selection menu screen and message library
#SDPRBLD	Services function driver build print page	#SDWKCTL	Subtask work controller
#SDPRCTL	Services function driver print control	#SDWKROU	Subtask work router
#SDPRCUP	Sends process destroyed message	#VLCCTL	Single VMC log control
#SDQCCRQ	Creates/extends/destroys queue	#VLDUMP	VMC log dump interface
#SDQCGAM	Gets/returns available message	#VLIPDES	Puts dump entry string to VMC log
#SDRICTL	Requests I/O control	#VLIPL	VMC log initial program load (IPL)/cleanup
#SDRIEXH	Handles exceptions during request I/O	#VLRGDES	Gets VMC log dump entry string from VMC log
#SDRIMIF	Requests a machine interface function	#VLRSLCT	VMC log retrieves/selects VMC log
#SDRIMIR	Handles machine interface response	#VLTASK	VMC log insert task
#SDSCCTL	Processes session type request I/O	#VL82EH	VMC log 82 exception handler
#SDSERVT	Services vector table		
#SDSMCRS	Creates/extends/destroys queue		
#SDSMGRP	Gets/frees real page		
#SDSSDVA	Varies on/off source/sink object		
#SDSSFND	Finds source/sink object		
#SDSSMOD	Modifies source/sink object		
#SDTCCRT	Creates service function driver tasks		
#SDTCDST	Destroys service function driver tasks and segments		



Authorization Management

INTRODUCTION

Authorization management controls the use of objects, system resources, and privileged machine instructions. This control is established by monitoring the following types of authorization.

- **Privileged Instruction:** Authorization to issue privileged machine instructions such as those that create and modify user profiles.
- **Resource:** The amount of auxiliary storage a user profile is authorized to allocate for its permanent objects.
- **Object:** Authorization to use system objects, ensuring that the objects referenced by an executing process are used only in the ways permitted.
- **Special:** Authorization to perform special operations on objects for which the executing process does not have specifically granted authorization (for example, objects that are being loaded).

The user profile is the collection point for authorization-related data. Every process in the system executes under control of a user profile, allowing both the system and the user to monitor and control the activities of each executing process. The information in the user profile sets limits to what can be performed by an executing process. Attempts by a process to exceed its authority result in exceptions and events.

Any user profile can be unique to one user or can be shared by several users. (User refers to external user of machine instructions, including but not limited to the system operator, security officer, IBM customer engineer, command work station operators, and others authorized to use the system.) Processes are initiated and run under control of a process user profile.

Authorization management also supports authorizations implied by adopted user profiles. At program creation, the program owner can specify that the user profile can be adopted by other users (user profiles) during subsequent uses of the program. This means that any object authorizations needed to successfully execute the program are adopted with the program. Thus, the user of a program with the adopted user profile attribute has an extended set of authorizations during execution of the program. When the program returns, the authorizations of the adopted user profile are no longer available to the process.

Authorization management supports the following machine instructions:

- Create User Profile
- Destroy User Profile
- Grant Authority
- Grant-Like Authority
- Materialize Authority
- Materialize Authorized Objects
- Materialize Authorized Users
- Materialize User Profile
- Modify User Profile
- Retract Authority
- Transfer Ownership
- Test Authority

Authorization management also provides the following functions:

- Retrieve authority information
- Test object authority
- Validate privileged instruction and special authorization
- Record and delete object ownership

Authorization Enforcement

Almost all VMC routines must check to ensure that the user that invoked a VMC routine is authorized to perform the requested operation. This check can be performed directly on the user profile by the VMC routine or by using the functions in authorization management. The checks that can be made on a user profile are:

Privileged Instruction: A user's rights to use privileged instructions are defined in the user profile. In general, the privileged instruction checking routine of authorization management is called by the VMC routine supporting the instruction to be executed. This routine verifies the user has the authority to perform the privileged instruction. The authorization management routine performs this verification by checking the bits that define the privileged instructions authorized in the adopted (if present) or the process user profiles.

Resource Usage: Auxiliary storage management accounts for the auxiliary storage space allocated to objects owned by the user profile. If the specified limit will be exceeded by a create or extend operation, storage management sets a return code. The calling VMC routine then signals a user profile storage limit exceeded exception. Auxiliary storage management optionally signals the exception.

Object Authorization: Authorization management (and #CFOCHKR) checks the authorization bits in the system pointer and the public authority bits in the object header; if sufficient authorization is not established with this check, authorization management then checks the adopted (if present) and process user profiles to verify the authorization requirements, and checks all object special authority in the user profile and the object authorization entries in the associated index. If sufficient authority cannot be established, authorization management signals (optionally) an authorization violation event and sets a return code; the calling VMC routine signals an exception.

Special Authorization: This authorization is checked by authorization management.

Recovery

There are requirements for maintaining valid user profiles and allowing normal operation, even if the user profile is damaged:

- Cleanup of dangling pointers whenever they are found in a user profile during materialize, transfer ownership, and other operations that modify authority.
- Saving all index updates on auxiliary storage immediately after modification.
- During object destruction, errors are logged and the operation continues to conclusion, but damaged or nonexistent user profiles are ignored.

DATA AREAS

User Profile

The user profile is a system object and is shown in Figure 14-1. A user profile is contained in three segment groups. The first segment group contains the following information:

- The segment group header
- The encapsulated program architecture (EPA) header
- Special, resource, and privileged instruction authorizations
- Recovery-related fields
- An index containing object authorizations

The EPA header of every object contains the following authorization management related information:

- Public authority (the authority of any process to access that object)
- A pointer to the owning user profile
- The object authority of the owner of the object

The following authorizations are contained in the user profile:

- Special Authorization
 - Implicit object authorizations
 - Machine attribute modification authorizations
- Privileged Instruction: Privileged instructions authorized to this user profile
- Resource Authorization: Amount of auxiliary storage allowed for the object owned by this user profile
- Utilization Data: Amount of auxiliary storage currently used by this user profile for permanent objects (this amount is updated by storage management during operations on permanent objects)

The user profile index contains pointers to the object owned by or authorized to this user profile. It also contains pointers to other user profiles that are authorized to use the objects owned by this user profile. The user profile index contains the following types of entries:

- Object Ownership Entry: Pointer to the object
- Authorized Object Entry: Pointer to the object and authorizations
- Authorized User Entry: Pointer to the object and a pointer to authorized user

The second segment group contains a table that has the same information (redundant) as the user profile index. This table is used for recovery purposes if the user profile index is damaged.

The third segment group contains the following information:

- The segment group header
- The associated space

Base Segment Group

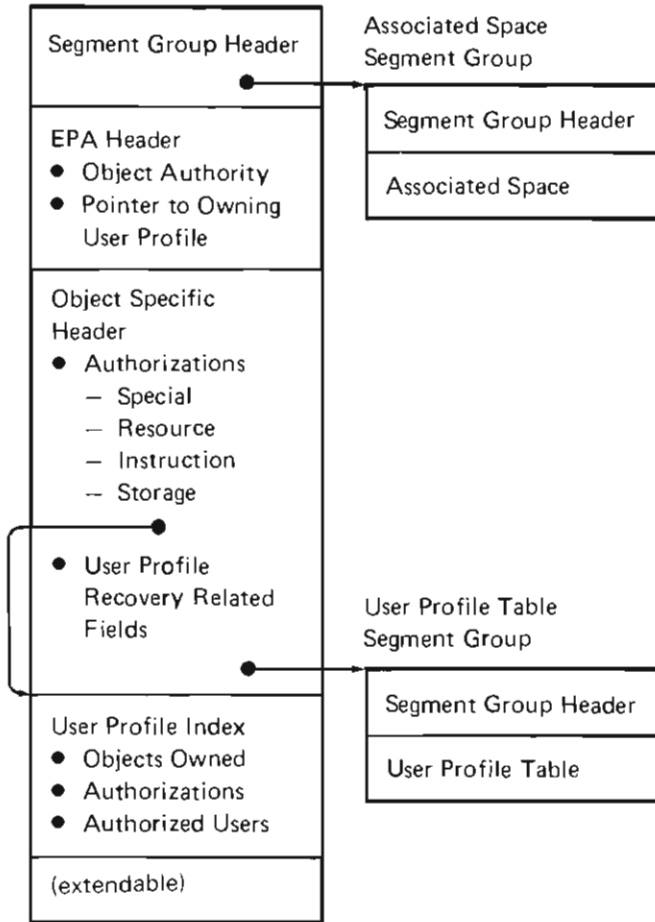


Figure 14-1. User Profile

System Pointer

The system pointer can contain bits describing the object authority to the object addressed by this pointer.

Process Control Block

The process control block (PCB) contains a pointer to the user profile governing its execution.

Invocation Control Block

The invocation control block contains a pointer to the adopted user profile (if any) governing its execution. Bits in the invocation control block indicate if a user profile is adopted and if it can be shared by other invocations. Adopted user profiles cannot be shared by external exception and event handlers.

User Profile Recovery

Auxiliary Storage Usage Field

There are a number of ways that this field may become inaccurate. Most of these involve some form of system crash in which either storage management shutdown is not performed or all processes are not brought to an instruction boundary before machine termination. The authority initialization module (#AUNIT), which runs during IMPL, validates the auxiliary storage usage field if either of the above conditions exist. #AUNIT first validates the total object size of each permanent object in the system. Then the #AUNIT places the sum of the object sizes for each object owned by a given user profile into the auxiliary storage usage field.

User Profile Index and Table

The information in the user profile index is stored, redundantly, in the user profile table. If either is determined to be damaged, it is rebuilt using the information in the undamaged part. If the damage is detected while accessing the information (**#AUINDEX**), the rebuild occurs dynamically by invoking the rebuild module (**#AUIXRBL**). If the rebuild is unsuccessful, the damage bit is set in the user profile.

Rebuild may also occur at IMPL time (**#AUIINIT**) if the user profile was being updated when a system crash occurred. Each time the index is to be updated, the index-in-use count in the object specific header is incremented and written to auxiliary storage. When the update is complete, the index-in-use count is decremented. If, at IMPL time, this count indicates that the user profile was in use when the machine terminated, the index is assumed to be damaged and is rebuilt. (If a table rebuild was in progress when the machine terminated, the table is rebuilt.)

STRUCTURE

The following is a list of the modules in authorization management and the function that each module performs. The list also shows how the module is invoked.

#AUCRTUP Create User Profile

Function: Creates a new user profile according to the input specifications.

How Invoked: Create User Profile instruction.

#AUCSEH Authorization Component-Specific Exception Handler

Function: Processes exceptions that occur within authorization management.

How Invoked: Through exception management.

#AUDESUP Destroy

Function: Destroys the specified user profile.

How Invoked: Destroy User Profile instruction.

#AUEXCEPT Generate Exception

Function: Generates authority-related exceptions and events.

How Invoked: Other VMC components.

#AUGRAU Grant Authority

Function: If public authority is specified, updates the target object EPA header; if private authority is specified, adds the authorized object entry to the receiver user profile and inserts an authorized user entry into the owning profile.

How Invoked: Grant Authority instruction.

#AUGRLAU Grant-Like Authority

Function: Performs the necessary grants to make a receive user profile look like a source user profile. The normal rules for materializing authorized users and for grant authority are in effect.

How Invoked: Grant-Like Authority instruction.

#AUIINIT Authority Initialization

Function: Initializes authority component and performs necessary recovery of damaged user profiles.

How Invoked: Other VMC components.

#AUINDEX Index Interface

Function: Provides the interface between authorization management and machine index management.

How Invoked: Authorization management modules that require index operations.

#AUIXRBL Index Rebuild

Function: Rebuilds the user profile index or table.

How Invoked: Other VMC components.

#AUXUTL User Profile Index Utility

Function: Initializes the user profile index.

How Invoked: Other VMC components.

#AUMATAU Retrieve Authority

Function: Retrieves the authorizations from a user profile and puts them into a specified area.

How Invoked: Materialized Authority instruction.

#AUMATOB Materialize Authorized Objects

Function: Retrieves and materializes the authorized object entries from a specified user profile.

How Invoked: Materialize Authorized Objects instruction.

#AUMATUP Materialize User Profile

Function: Retrieves and materializes the instruction, special, and resource authorizations from a user profile.

How Invoked: Materialize User Profile instruction.

#AUMATUU Materialize Authorized User

Function: Searches the profile index of the owner of the object for a pointer to authorized users, and retrieves the authorized user names from the profile header.

How Invoked: Materialize Authorized Users instruction.

#AUMODUP Modify User Profile

Function: Modifies the authorizations in a user profile.

How Invoked: Modify User Profile instruction.

#AURCTAU Retract Authority

Function: For public authorization, inserts new authorizations into the EPA header of the target object; for private authorization, removes the authorized object entry from the user profile of the receiver.

How Invoked: Retract Authority instruction.

#AUTBUTL User Profile Table Utility

Function: Creates, validates, and extends the user profile table.

How Invoked: Other VMC components.

#AUTSTAU Test Authority

Function: Tests the authorized entry in the user profile of the receiver.

How Invoked: Test Authority instruction.

#AUVERCH Version Change

Function: Performs changes to the user profile to convert to the release 2 format.

How Invoked: Other VMC components.

#AUWCHEK Segment Identification Wrap Check

Function: Removes references to nonexistent objects from a user profile.

How Invoked: Other VMC components.

#AUXCHEK User Profile Crosscheck

Function: Checks all user profiles for consistency.

How Invoked: Other VMC components.

#AUXFRO Transfer Ownership

Function: Updates the object header to reflect new owner, transfers ownership and authorized user entries from old owning profile to the user profile of the new owner.

How Invoked: Transfer Ownership instruction.

#CFAUTH Authorization Common Function

Function: Validates the privileged instruction and special authorizations (#CFAUPRV); retrieves the authorizations of a specified object (#CFAURET); tests user profile to determine if a user has the authority to execute the operation requested (#CFAUTST); and retrieves the object authority available to a process (#AUTSTAU).

How Invoked: Test Authority instruction and from other authorization management modules that require authorization verification.

#CFDELOO Delete Object Ownership

Function: Deletes ownership of a given object from the owning user profile.

How Invoked: Other VMC components.

#CFRECOO Record Object Ownership

Function: Records ownership of a given object in a given user profile.

How Invoked: Other VMC components.



Context Management

INTRODUCTION

Contexts are used to store addressability to system objects. The user creates contexts, inserts or deletes addressability into or from contexts, and transfers addressability from one context to another. These functions are accomplished by using System/38 instructions. Addressability to a system object can only be in one context at a time. Addressability to an object need not be kept in a context, but the user must keep a system pointer to the object in order to maintain addressability to the object.

Once a system object is addressed by a context, the object can be located in virtual storage by using the Resolve System Pointer instruction or implicitly by using an unresolved pointer. Using unresolved pointers is called late binding, and is described in more detail in the *Program Management* section of this manual.

Context management supports the following System/38 instructions:

- Create Context
- Destroy Context
- Materialize Context
- Modify Addressability
- Rename System Object
- Resolve System Pointer

Context management also supports the following functions:

- Check dangling pointer
- Find entry in context (indirect support)
- Find entry in a name resolution list (NRL) context (indirect support)
- Insert context entry
- Delete context entry
- Resolve system pointer

Data Pointer Resolution

Data pointers are resolved either explicitly by using the Resolve Data Pointer instruction or implicitly by using an unresolved data pointer. The Resolve Data Pointer instruction results in supervisor linkage (SVL) to #MNRESLVD. An unresolved data pointer results in the verify exception handler (#CFVFEH) being invoked. #CFVFEH in turn invokes #MNRESLVD at entry point #CFRESOD. In either case, #MNRESLVD returns a pointer to the data object if one exists.

Recovery

The primary recovery considerations are:

- The ability to handle a damaged context
- The ability to handle dangling pointers
 - Context entries pointing to nonexistent or invalid object
 - Objects pointing to a context that does not exist
- Support of damage tolerant destroy system object functions

A damaged context is detected by context management as a result of a return code from an index control block operation by index management. The context is then marked as damaged. A destroy context operation is the only context management function that can be performed on that context marked as damaged.

Dangling pointers are handled by additional checks:

- Whenever a context entry is examined, an optional check can be made to determine that the context addresses the proper object (the segment extender, object name, and the back-pointer to the addressing context are verified).
- Whenever the context pointer from an object is used, a check is made to determine that the pointer addresses an existing context.

DATA AREAS

Contexts

There are three types of contexts:

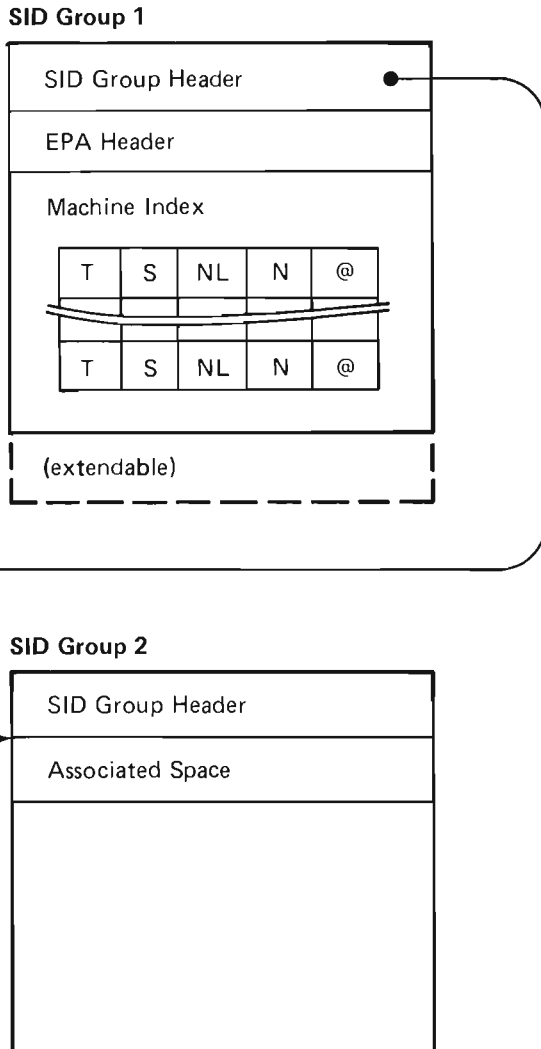
- **Machine Context:** This context is built at installation time by the context rebuild function and is permanently assigned a unique virtual address. This address can be found at label #MCA4VMC in the machine communication area (MCA). The context rebuild function rebuilds the machine context if it is damaged or destroyed. This includes locating all permanent objects that must be addressed by the machine context and reinserting them in the machine context. The context rebuild function also rebuilds all permanent contexts that are damaged and marked as eligible to be rebuilt. The machine context contains addressability to all of the following objects and cannot contain addressability to any other object:
 - Logical unit descriptions
 - Controller descriptions
 - Network descriptions
 - User profiles
 - Permanent contexts

- **Permanent Contexts:** These contexts are built as a result of Create Context instructions. Permanent contexts can contain addressability to any object (permanent or temporary) except those addressed by the machine context or those that cannot be addressed by any context (temporary contexts, for example).
- **Temporary Contexts:** These contexts are also built as a result of Create Context instructions, and contain addressability to the same types of objects as permanent contexts. Because addressability to temporary contexts is not kept in the machine context, the user must maintain addressability to a temporary context in a system pointer. If a temporary context is not explicitly destroyed by the user, it is destroyed by VMC as part of machine processing termination.

Contexts reside in one or two segment groups as shown in Figure 15-1. The first segment group contains a segment group header, an encapsulated program architecture (EPA) header, and a machine index that contains the actual context entries.

The machine index portion of a context contains variable length entries in the following form:

T	S	NL	N	@
(1)	(1)	(1)	(*)	(8)



where:

- T identifies the *type* of the system object addressed
- S is a user-defined qualifier (*subtype*)
- NL identifies the *name length*, with trailing blanks removed to reduce context space requirements
- N is the user specified *name* of the object
- @ is the 8-byte *address* of the EPA header of the object

Note: The format of the context entries in the machine index is different from the format seen by the user.

Figure 15-1. Encapsulated Context Format

Figure 15-2 shows the relationship of contexts, name resolution list, and other pointers in the system. Some relationships are described as follows:

- A A context pointing to another object
- B NRL pointing to contexts
- C A pointer to the context in the EPA header of an object addressed by that context
- D Object not addressed by a context

Figure 15-2 also shows exception conditions that can exist because of some unusual condition such as a system failure. The conditions are:

- E A context entry contains a pointer to an object that does not exist.
- F The EPA header of an object contains a pointer is made up of all 0's or a pointer to a context that does not exist.

These conditions, called dangling pointers, are allowed in the system. E occurs if an object is destroyed but the context entry was not deleted. This is the case after a system failure for permanent contexts that address temporary objects. F occurs if a context that contains addressability to one or more system objects is destroyed.

For rename object operations, E is also considered to be dangling if the type, subtype, and name in the object EPA header and the entry in the context do not match.

Type E dangling pointers are handled in the following ways:

- For inserts, the address portion of the entry is replaced with the new entry address.
- For deletes, there is no effect on the operation.
- For resolves, dangling pointers are ignored.

Type F dangling pointers are handled depending on the function being performed. Functionally, the object is considered not to be addressed by a context.

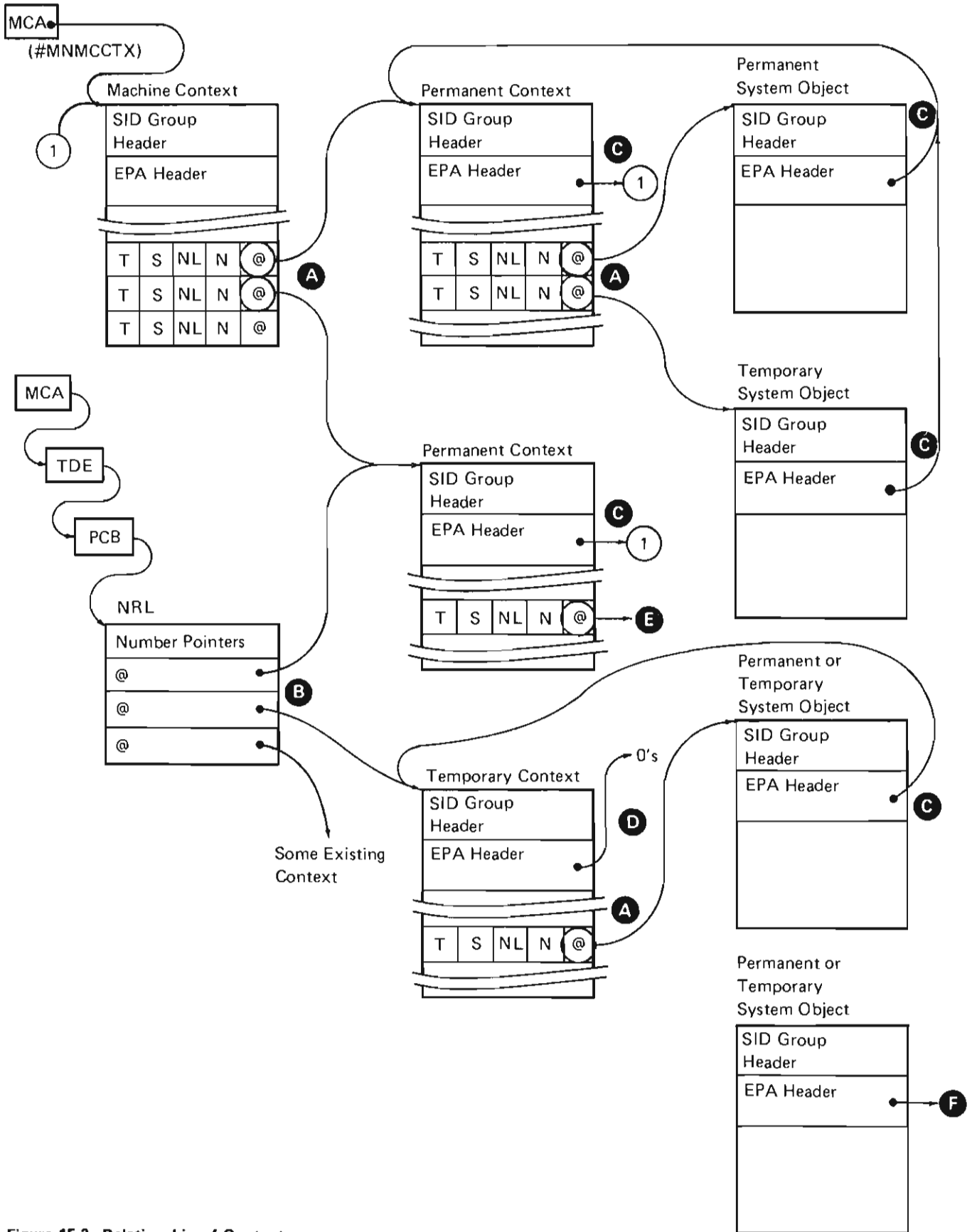


Figure 15-2. Relationship of Contexts

Name Resolution List

The NRL is a portion of a space that contains one or more resolved system pointers to contexts. The contexts contained in the NRL are selected by the user. The NRL is supplied as an attribute of the PDT used by the Initiate Process instruction. The NRL can be altered by using the Modify Process Attributes instruction. The NRL is used to specify the contexts to be searched and the sequence of the search when attempting to resolve a system pointer. System pointers in the NRL must be resolved. The format of the NRL is shown in Figure 15-3.

Number Pointers	Reserved
System Pointer to Context	
System Pointer to Context	
System Pointer to Context	

Figure 15-3. Name Resolution List

Encapsulated Program Architecture Header

The EPA header is part of an object used to describe that object. The header contains the type, subtype, name, and if the object is addressed by a context, a pointer to that context.

Machine Communication Area

The MCA is used to map storage areas and provide control areas. The MCA contains a pointer to the machine context and TDE.

Process Control Block

The process control block (PCB) is used to map storage areas and provide control areas. The PCB contains a pointer to the NRL.

Task Dispatching Element

The task dispatching element (TDE) is used to map storage areas and provide control areas. The TDE contains a pointer to the PCB.

STRUCTURE

The following is a list of the modules in context management and the function that each module performs. The list also shows how the module is invoked.

#CFDELCE Delete Object Addressability

Function: Deletes addressability to an object from the specified context.

How Invoked: Other VMC components.

#CFINSCE Insert Object Addressability

Function: Inserts addressability to an object into the specified context.

How Invoked: Other VMC components.

#CFRESSP Resolve Pointer

Function: Resolves a late-bound (unresolved and initialized) system pointer.

How Invoked: Other VMC components.

#MNCRTC Create Context

Function: Creates a context with the attributes specified and returns addressability to the created context.

How Invoked: Create Context instruction.

#MNDGPCK Check if Context Entry is Dangling

Function: The specified context entry is expanded, the entry is checked to determine if it is dangling, and the appropriate information is returned.

How Invoked: Within this component.

#MNDSTC Destroy Context

Function: Destroys the specified context.

How Invoked: Destroy Context instruction.

#MNMATC Materialize Context

Function: Materializes selected information from a context.

How Invoked: Materialize Context instruction.

#MNMODA Modify Addressability

Function: Inserts addressability into a context, deletes addressability from a context, or transfers addressability from one context to another.

How Invoked: Modify Addressability instruction.

#MNRCTX Resolve–Find Object in One Context

Function: Attempts to locate the proper entry in the specified context and returns the appropriate information.

How Invoked: Within this component.

#MNRENAM Rename System Object

Function: Changes the symbolic identification (name, subtype, or both) of a permanent or temporary system object, and updates any contexts that reference that object.

How Invoked: Rename Object instruction.

#MNRESSP Resolve System Pointer Control Module

Function: Resolves a system pointer, and if specified, sets or modifies authority.

How Invoked: Resolve System Pointer instruction.

#MNRNRL Resolve–Find Object in NRL

Function: Searches the NRL contexts to locate the specified object entry and returns the appropriate information.

How Invoked: Within this component.

#MNRSLVD Resolve Data Pointer Addressability

Function: Resolves a data pointer to the address and attributes of a data pointer.

How Invoked: Other VMC components.



Recovery Initialization

INTRODUCTION

Recovery initialization is a recovery common function used by VMC components to recover objects at IMPL time. An overview of the recovery initialization function is shown in Figure 16-1. Recovery Initialization performs the following functions:

- Builds a machine index with an entry for every base segment of an object and every secondary segment for multiple segment objects.
- Builds an index containing every user profile for use by authority initialization.
- Provides an interface to retrieve selected entries from the object segment index.

#RCINIT is invoked by the first IPL recovery function that needs to recover objects and create the send/receive tasks. #RCINIT then finds an entry in the permanent directory and sends that entry to the task reading the storage unit for that permanent directory entry. #RCBBSIX, running under each task, reads the disk and puts an entry in the object segment index for that segment. If the segment read is a user profile, an entry is put in the user profile index.

#RCREAD is invoked to retrieve entries from the object segment index and at the same time, the object type or secondary segment type for the next object segment to be returned from the index can be specified.

The recovery read function protects the user from damage to either the base segment index or the object(s) being returned. If a find is attempted to the base segment index and the base segment index is found to be unusable, it is rebuilt. If the rebuild function fails, machine processing terminates with a machine check. The read function assures the segment header that the segment returned is readable at the time the base segment index is built.

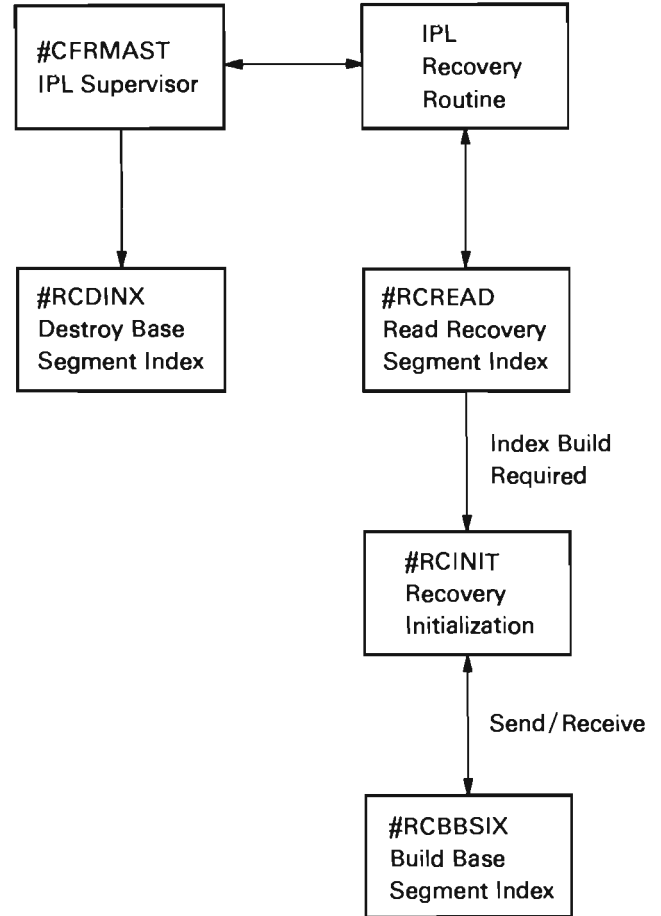


Figure 16-1. Recovery Initialization Overview

STRUCTURE

The following is a list of the modules in recovery initialization and the function that each module performs. The list also shows how the module is invoked.

#RCBBSIX Build Base Segment Index

Function: Searches the storage management permanent directory and builds the following:

- An index containing the address of the base segment, secondary segments, and associated space of each permanent object in the system.
- An index containing the address of each user profile in the system.

How Invoked: Within this component.

#RCDINX Destroy Recovery Segment Index

Function: Destroys the recovery segment machine index.

How Invoked: Other VMC components.

#RCINIT Recovery Initialization

Function: Initializes the recovery component and builds the data areas for use by #RCBBSIX.

How Invoked: Other VMC components.

#RCREAD Read Recovery Segment Index

Function: Selects entries from the machine index of the system object segments.

How Invoked: Other VMC components.

Program Execution Management

INTRODUCTION

Program execution management provides the linkage between programs and between programs and subroutines within the programs. Program execution management performs the following basic functions:

- Activates a program: Puts a program into a ready-to-execute state
- Invokes a program: Causes the program to be executed
- Modifies storage allocation: Extends or truncates the allocated process automatic storage area, and extends the allocated process static storage area
- De-activates a program: Removes the program from an executable state
- Destroys a program invocation: Terminates the execution of a program

These functions provide support for the following System/38 instructions:

- Activate Program
- Call External
- Call Internal¹
- De-activate Program
- End
- Modify Automatic Storage Allocation
- Return External
- Set Argument List Length¹
- Store Parameter List Length¹
- Transfer Control

Program Activation

All programs executing in the system operate under control of a process. Before a program can be activated and executed, a process under which that program is to execute must be established. Process management performs the process creation operation and initially allocates the areas used by the process. Process creation is described in the *Process Management* section of the manual.

¹Support for these instructions is provided by code that is generated by the translator and inserted into the instruction stream of the encapsulated program.

Once a process has been established, program execution management can activate a program. An Activate Program instruction initiates the activation operation. The primary object of the activation operation is to initialize the process static storage area (PSSA) of the program. If a program does not contain any static areas, the program is considered permanently activated and does not require activation.

Program Invocation

The invocation function of program execution management controls the synchronous execution of programs within a process. The invocation function allows control to be passed from one program instruction stream to another, and allows for a subsequent return of control when a function is complete.

Programs can be invoked under the following conditions:

- A process definition can specify a program to be invoked as part of the process initiation phase.
- A process definition can specify the first program to be invoked in the problem phase. When process initiation enters the problem phase, the first process problem state is given control.
- A Call External instruction causes execution of the invoking program to be suspended and control to be passed to the referenced program.
- A Transfer Control instruction causes execution of the invoking program to be suspended, the invocation of the invoking program to be destroyed, and control to be passed to the referenced program.
- 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 data space index can specify a user exit program to invoke.
- A process definition can specify a program to be invoked as part of the process termination phase.

Invoking a program causes the following to occur:

- Execution of the invoking program is suspended and the current status of the invoking program is saved pending return of control.
- An invocation entry for the program is allocated in the process automatic storage area (PASA). This entry contains an allocation for each object that has the automatic allocation attribute.
- The automatic objects are assigned their initial values.
- Exception descriptions that are defined in the program are activated to process the associated exceptions.
- Parameter objects (if any are defined in the invoked program) are resolved to argument objects passed by the invoking program.
- Authority of the program being invoked is verified.
- The program is implicitly activated if it was not previously activated.
- Control is passed to the entry point defined in the instruction stream of the program to be invoked. Instruction execution continues in the invoked program until an invocation of another program is encountered or the end of the program is reached.

Program De-activation

An activation entry can be marked as not active using the De-activate Program instruction. An activation entry that is not active must be activated before it can be used in an invocation. The system implicitly reactivates an inactive entry when the associated program is invoked.

Invocation Destruction

When an invoked program relinquishes control, the associated invocation is deallocated and the following operations occur:

- Execution of the invoked program is suspended.
- The automatic space is deallocated.
- The exception descriptions associated with the invocation are made inactive.
- An invocation exit program set for the invocation being destroyed is optionally invoked if the invocation is destroyed because of a return from exception or a signal exception.
- Control is passed to some point in a previously invoked program.

The invocation relinquishes control and is subsequently destroyed under the following conditions:

- Return external: The invocation is destroyed and control is passed to the invocation immediately preceding the destroyed invocation in the process chain.
- Transfer control: The invocation is destroyed and the target program of the Transfer Control instruction is invoked.
- Return from exception: The exception handling sequence returns control to a previous invocation.
- Signal exception: The exception presentation sequence passes control to a previous invocation.

DATA AREAS

Process Automatic Storage Area

The contents of the PASA are shown in Figure 17-1. The PASA starts on a quad-word boundary and begins with a 96-byte control element that contains the following information:

- A pointer to the newest PASA element
- A pointer to the first PASA element
- A pointer to the next available PASA element
- The invocation mark counter (4 bytes)

Following the control element are the PASA elements. Each element starts on a quad-word boundary and contains the following information:

- A pointer to the previous PASA element
- A pointer to the next PASA element if one exists
- A pointer to the program associated with this PASA element
- The current invocation count
- The type of the associated invocation
- The invocation mark count
- The user area
- The automatic data area

The invocation control block addresses the associated PASA element so that when an invocation terminates and the invocation is removed from the invocation work area, the associated PASA element is also removed from the chain.

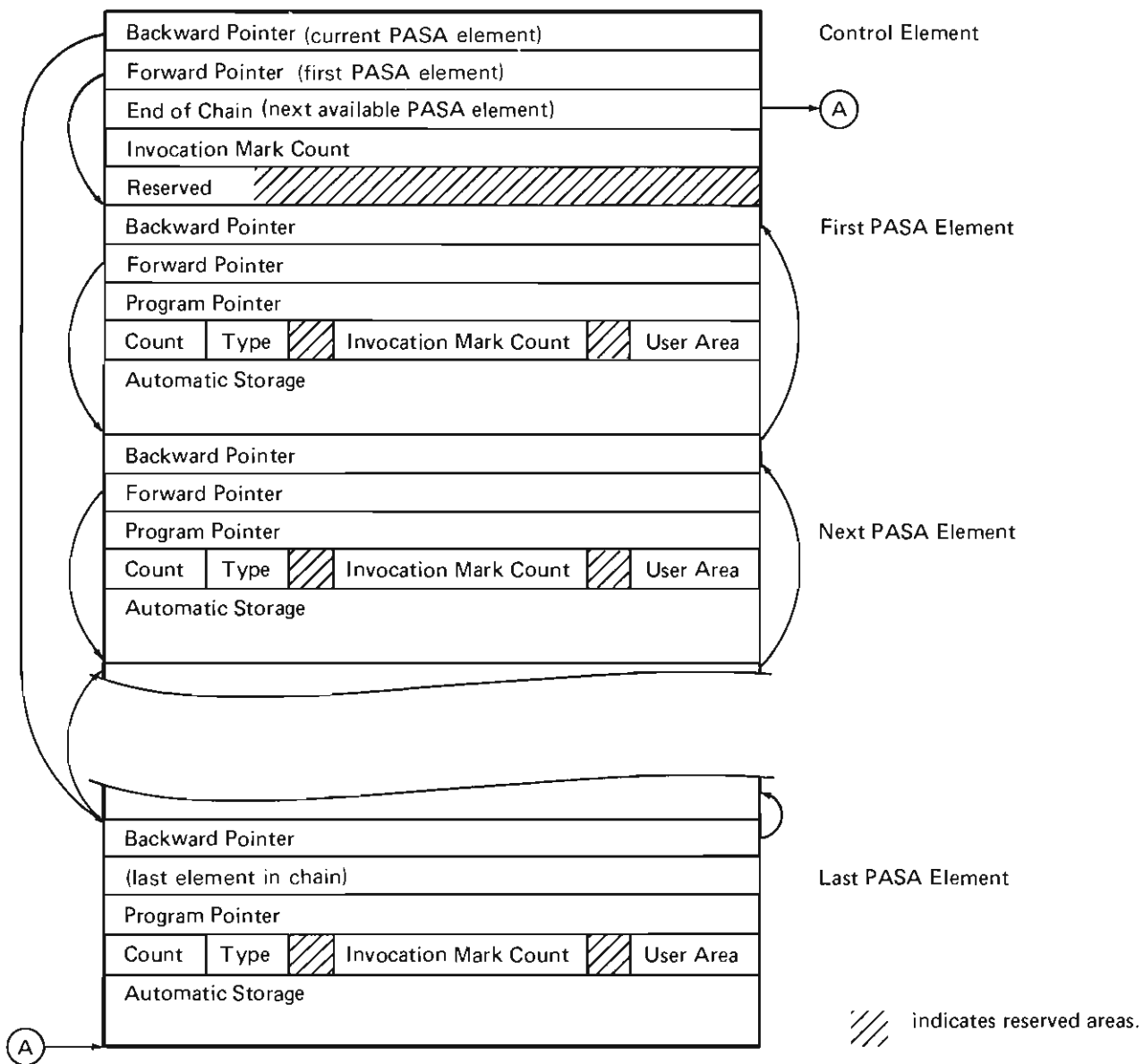


Figure 17-1. Process Automatic Storage Area

Process Static Storage Area

The contents of the PSSA are shown in Figure 17-2. The PSSA starts on a quad-word boundary and begins with a 96-byte control element that contains the following information:

- A pointer to the newest element on the chain
- A pointer to the first element on the chain if one exists
- A pointer to the next available PSSA element
- The PSSA modification flags

Following the control element are the PSSA elements. These elements exist for programs that require static storage. The elements can be created either explicitly by using an Activate Program instruction or implicitly by using Call External and Transfer Control instructions. Each PSSA entry starts on a quad-word boundary and contains the following information:

- A pointer to the previous PSSA element
- A pointer to the next PSSA element if one exists
- A pointer to the associated program
- The activation count
- The active flag
- The invocation count
- The activation mark count
- The length of this PSSA entry
- The static area for the program

Unlike invocation control block and PASA elements, PSSA elements are not associated with an invocation and are not destroyed (removed from the chain) when an invocation terminates. Space associated with de-activated PSSA elements is reused by the machine when a new PSSA element is created.

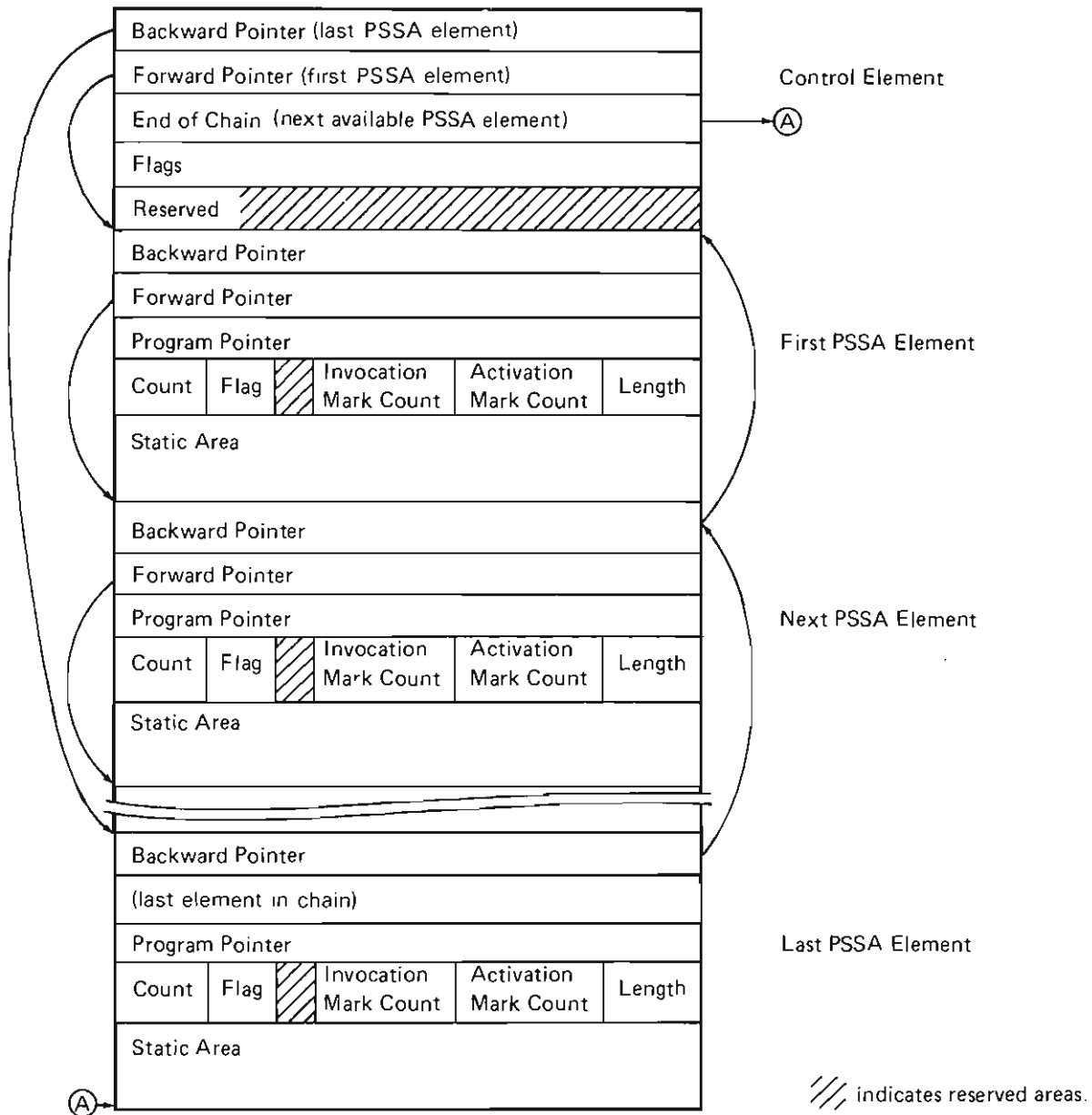


Figure 17-2. Process Static Storage Area

STRUCTURE

The following is a list of the modules in program execution management and the function that each module performs. The list also shows how the module is invoked.

#AICALLM Transfer Control

Function: Transfers control from one program to another.

How Invoked: Transfer Control instruction or other VMC components.

#AICALLX Call External

Function: Passes control to a user program invocation. PSSA and PASA entries are created and initialized as required.

How Invoked: Call External instruction.

#AICMACH Call from Machine

Function: Invokes a program from the machine.

How Invoked: Other VMC components.

#AICRACT Create Activation

Function: Initializes PSSA for a program.

How Invoked: As a result of one of the following:

- Activate Program instruction
- From **#AICALLX** or **#AICALLM** resulting from an implicit activation at program invocation

#AIDACTV De-activate Program

Function: De-activates a program.

How Invoked: De-activate Program instruction.

#AIMDASA Automatic Storage

Function: Changes the automatic storage size for the current invocation.

How Invoked: Modify Automatic Storage Allocation instruction.

#AIRTXT Return External/End

Function: Returns control to the next previous invocation.

How Invoked: Return External and End instructions.



Program Management

INTRODUCTION

Program management performs the following functions:

- Translates (encapsulates) a program template into an executable object
- Provides addressability to the encapsulated object
- Materializes the attributes of the object
- Deletes addressability to the object

Program management supports the following System/38 instructions:

- Create Program
- Materialize Program
- Destroy Program
- Delete Program Observability

Program Creation

Before a program can be executed in the system, program management must convert the program into an executable form. The process of converting (called encapsulation) the program into executable form is initiated by a Create Program instruction, and the conversion process is performed by the translator (#XLATOR).

Because the operation of the translator is similar to that of a compiler, the internal operation of the translator is described only in general. The input to and the output from the translator are emphasized in this description.

The input to the translator is the program template, the output from the translator is an encapsulated program system object. The program template contains a description of the program to be created. This information is as follows:

- Template header that contains the creation attributes of the program and pointers to the remainder of the template
- The object definition table (ODT) that contains the following:
 - The ODT directory vector (ODV) containing entries that described the objects used by the program and a pointer to an ODT entry string if the descriptions of the objects cannot be completely contained in the ODV
 - The ODT entry string that contains variable length entries that complete the definitions of the object not completely described in the ODV
- The instruction stream of the program

The program template is built by a program resolution function of either the control program facility (CPF) or other control program.

The encapsulated program system object contains the following:

- The encapsulated program architecture (EPA) header that contains offsets and sizes relating to the object
- The internal microprogramming instructions to be executed
- Static initialization code that contains constant values and microprogramming instructions used to initialize program static data
- Automatic initialization code that contains microprogramming instructions used to initialize program automatic data
- The object specific header that contains attributes, offsets and sizes relating to the program
- The object mapping table that contains entries that provide location mapping for each object defined by an ODV entry
- The external object list that contains the list of externally known names such as the external entry point into the program
- The exception directory that defines the exceptions to be handled for this program and pointers to the exception handlers for these exceptions
- User extendable space
- Materialization definition template that contains a copy of the input program template if the program was specified as observable in the Create Program instruction
- The break offset mapping table consisting of bit entries that map the start of all microprogramming instructions in the encapsulated program

Figure 18-1 shows an example of a user procedure, the program template for that procedure, and the resulting encapsulated program.

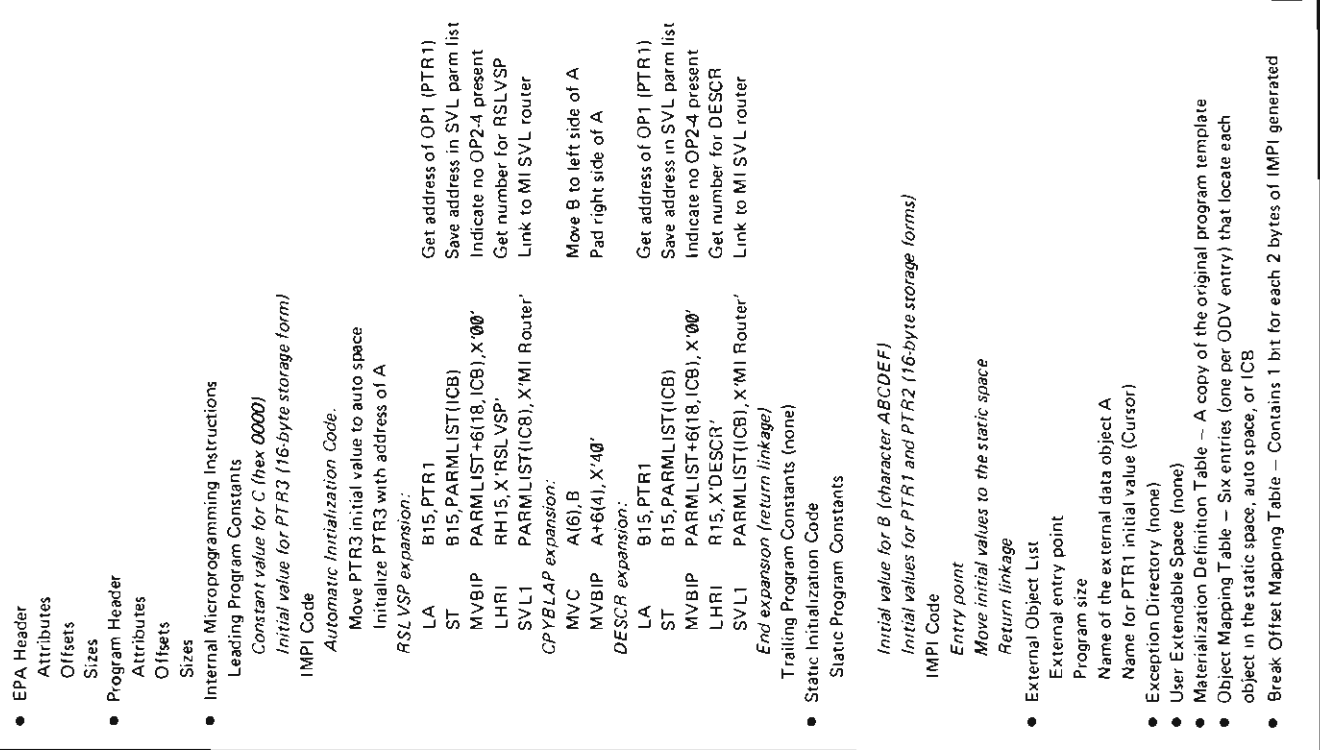


Figure 18-1. Sample Program

In general, the translator performs the following functions:

- Accepts the program template as input data
- Checks the input for proper syntax
- Generates a program system object
- Returns diagnostic information
- Returns a system pointer to the generated system object

The translator initializes an internal communication area and then invokes the following phases to perform the encapsulation procedure:

- Initialization (**#XINIT**) that validates the program template header and builds the materialization definition template
- ODT scan (**#XODTSC**) that checks the ODT for proper syntax and builds the external object list and the exception directory
- Instruction stream scan (**#XSCAN**) that checks the instruction stream for proper syntax and builds areas used by other translator phases
- Register optimization (**#XREGOPT**) that assigns addresses to registers, optimizes register assignments, and builds the register assignment chain
- Data generation (**#XDATAGN**) that builds the object mapping table (OMT), initial constants for static and automatic values, and a table for automatic storage initialization
- Code generation (**#XCODEGN**) that builds the internal microprogramming instruction (with optional constants), the automatic initialization code, the static initialization code, and the break offset mapping table
- Termination (**#XTERM**) that completes the program header and completes the encapsulation process

Exceptions detected during encapsulation are processed by the translator exception handler (**#XEH**).

Program Materialization

The basic attributes of an encapsulated program can be materialized. If the program was designated as observable when the program was encapsulated and the materialization definition template has not been destroyed, a copy of the program template can also be materialized along with the basic attributes.

The Materialize Program instruction causes program management module **#PGMATPG** to be invoked. This module materializes the attributes and the materialization definition template (if present) of the specified program into the area specified by the user. The materialization definition template contains a copy of the program template that is used to create the program and the OMT that is generated by the translator.

Program Destruction

An encapsulated program can be destroyed and addressability to the program can be removed by executing a Destroy Program instruction. This instruction causes program management module **#PGDESPG** to be invoked. This module verifies that the program is eligible for destruction then deletes the addressability to the program.

Program Observability

A program is observable if the encapsulated program contains a materialization definition template. The Delete Program Observability instruction causes program management module **#PGDELPO** to be invoked. This module deletes the materialization definition template and designates the program as not observable.

DATA AREAS

Program Template

The program template is the input to the translator. The template consists of the following:

- The template header that contains the creation attributes of the program and addressability to the remainder of the template.
- The instruction stream that contains the operations to be performed by the program. The instruction stream contains the following information for each instruction in the program:
 - Instruction operation code
 - Operation code extender field, if required
 - Instruction operand(s) if required
- The ODT that defines the amount and attributes of the storage to be allocated for the objects referenced in the program. The ODT contains the following components:
 - The ODV consisting of 4-byte entries. These entries either completely describe the attributes of an object or specify an offset into the ODT entry string where a complete description of the object can be found. Instruction operands that explicitly reference an object contain index values into the ODV.
 - The ODT entry string that completes the descriptions of those objects that cannot be completely described in an ODV entry.
- The break offset mapping table is an optional part of the program template. The break offset mapping table provides mapping of the System/38 instructions to the high-level language source statements. The translator does not use the break offset mapping table.
- The symbol table is an optional part of the program template. This table provides mapping of the objects referenced in the program to the high-level language source statements. The translator does not use the symbol table.

Encapsulated Program

Figure 18-2 shows the basic structure of an encapsulated program. The program consists of two, three, or four segment groups. Addressability to the program is to the first byte of the segment group header. The segment group header provides addressability to the remainder of the program, either directly, or indirectly through the object specific header. The segment groups can be either permanent or temporary, and can exist in an access group.

The contents of the first segment group are as follows:

- The segment group header
- The EPA header
- The object specific header for the program

- The internal microprogramming code that consist of the following:
 - Leading program constants: Constants at the beginning of the instruction stream used to initialize automatic storage or reference objects defined as constants in the ODT. These constants start on a doubleword boundary, have a maximum length of 65 535 bytes, and cannot cross a segment boundary.
 - Automatic initialization code: Instructions generated by the translator, used to initialize automatic storage. The code starts on a halfword boundary, has a maximum length of 65 535 bytes, and cannot cross a segment boundary. The automatic initialization code is included as part of the entry point instructions if it fits within the first segment of the program; otherwise, it is placed following the static initialization code in another segment.
 - Internal microprogramming instructions: Instructions generated by the translator that perform the functions of the System/38 instruction set. The code starts on a doubleword boundary has a maximum length of 1024 K bytes (including leading and trailing constants) and can cross segment boundaries. This code, generated for a particular System/38 instruction, can be either inline or supervisor linkage to another routine that performs the function.
 - Trailing program constants: Constants at the end of the instructions that are used by expansions to perform unique functions. This area is optional, starts on a doubleword boundary if present, has a maximum length of 4 K bytes, and cannot cross a segment boundary.

- The static initialization code that consists of the following:
 - Static program constants: Constants similar to those in the instruction stream except that these are used to initialize static storage.
 - Internal microprogramming instructions: Instructions generated by the translator, used to initialize static storage.

The static initialization code starts on a doubleword boundary, has a maximum length of 65 535 bytes, and cannot cross a segment boundary.

- The external object list that contains the following:
 - The external entry point of the program
 - Program length
 - Named external data objects
 - Initial value information for system and data pointers
- The exception directory that defines the exceptions that the program can handle and pointers to the exception handlers for those exceptions.

The second segment group is present only if the program is observable. This group contains the following:

- The segment group header.
- The materialization definition template that is a copy of the input program template.
- The OMT that consists of a variable-length vector of 6-byte entries. The number of entries correspond to the number of ODV entries. Each OMT entry provides a location mapping for the object defined by an associated ODV entry.

The third segment group contains the following:

- The segment group header
- The break offset mapping table that provides mapping to the System/38 instructions in the program template to the microprogramming instructions in the encapsulated program

The fourth segment contains the following:

- The segment group header
- The user space

Segment Group Header (YYSGHDR)
EPA Header (YYEPAHDR)
Object Specific Header (XPGMHDR)
<ul style="list-style-type: none"> • Internal Microprogram Instruction Stream • Automatic Initialization Code (included if space available in first segment)
Static Initialization Code
Automatic Initialization Code (if no space available in first segment)
External Object (YYEOLELT)
Exception Directory

Segment Group Header
Materialization Definition Template
Object Mapping Table

Segment Group Header
Break Offset Mapping Table

Segment Group Header
User Space
(extendable)

Figure 18-2. Encapsulated Program

STRUCTURE

The following is a list of the modules in program management and the function that each module performs. The list also shows how the module is invoked.

#PGDELPO Delete Observability

Function: Deletes the materialization definition template and designates the program as not observable.

How Invoked: Delete Program Observability instruction.

#PGDESPG Destroy Program

Function: Destroys addressability to the specified program.

How Invoked: Destroy Program instruction.

#PGMATPG Materialize Program

Function: Materializes the selected attributes of the specified program.

How Invoked: Materialize Program instruction.

#XCODEGN Code Generation

Function: Performs the code generation phase of a create program operation.

How Invoked: From #XLATOR.

#XDATAGN Data Generation

Function: Performs the data generation phase of a create program operation.

How Invoked: From #XLATOR.

#XEH Translator Component-Specific
Exception Handler (CSEH)

Function: Processes exceptions detected in a create program operation.

How Invoked: From #XLATOR.

#XINIT Initialization

Function: Performs common setup functions for a create program operation.

How Invoked: From #XLATOR.

#XLATOR Translator

Function: Performs common setup operations for a create program operation and invokes the translator phases.

How Invoked: Create Program instruction.

#XODTSC ODT Scan

Function: Validates the object definition table.

How Invoked: From #XLATOR.

#XREGOPT Register Optimization

Function: Optimizes register assignments.

How Invoked: From #XLATOR.

#XSCAN Instruction Stream Scan

Function: Validates the input instruction stream.

How Invoked: From #XLATOR.

#XTERM Termination

Function: Completes a create program operation.

How Invoked: From #XLATOR.

Advanced Program-To-Program Communications Station I/O Manager

INTRODUCTION

The link protocol characteristics for advanced program-to-program communications (APPC) station I/O manager may be either primary or secondary synchronous data link control (SDLC) and are determined when the network description (ND) object is created.

Under APPC station management, logical unit descriptions (LUDs) take on a new definition. APPC LUDs do not represent devices; instead, an APPC LUD represents a group of paths to another processor. A group of these are attached to a CD and represent a set of parallel, independent paths to another processor. These paths are referred to as assignable sessions. This means that an APPC LUD has no device-unique data. The APPC LUD, however, contains SNA information. When a program opens a file using APPC LUD, 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 communications files under the same or other processes.

The APPC station IOM interfaces with the following:

- Machine services control point (MSCP)
- Primary/secondary SDLC IOM
- Error log
- Modify Controller Description instruction processor
- Modify Logical Unit Description instruction processor
- Request I/O instruction processor

The user of the APPC station IOM can execute Modify Controller Description and Modify Logical Unit Description instructions, and can make, break, and manage SNA paths to an LU within the other unit. The Request I/O instruction is used to communicate with and control sessions within the LU. The APPC station IOM handles the logical paths for each LU to which the System/38 is connected. An overview of the APPC station IOM is shown in Figure 19-1.

The APPC station IOM is a VMC task that is created by MSCP when a Modify Controller Description (vary-on) instruction is issued against an APPC controller description (CD) object. The task is created with one input send/receive queue upon which the send/receive messages are placed. The MSCP also provides the APPC station IOM with the address of the input send/receive queue of the line IOM that will be used and with the address of a controller description block for exception handling.

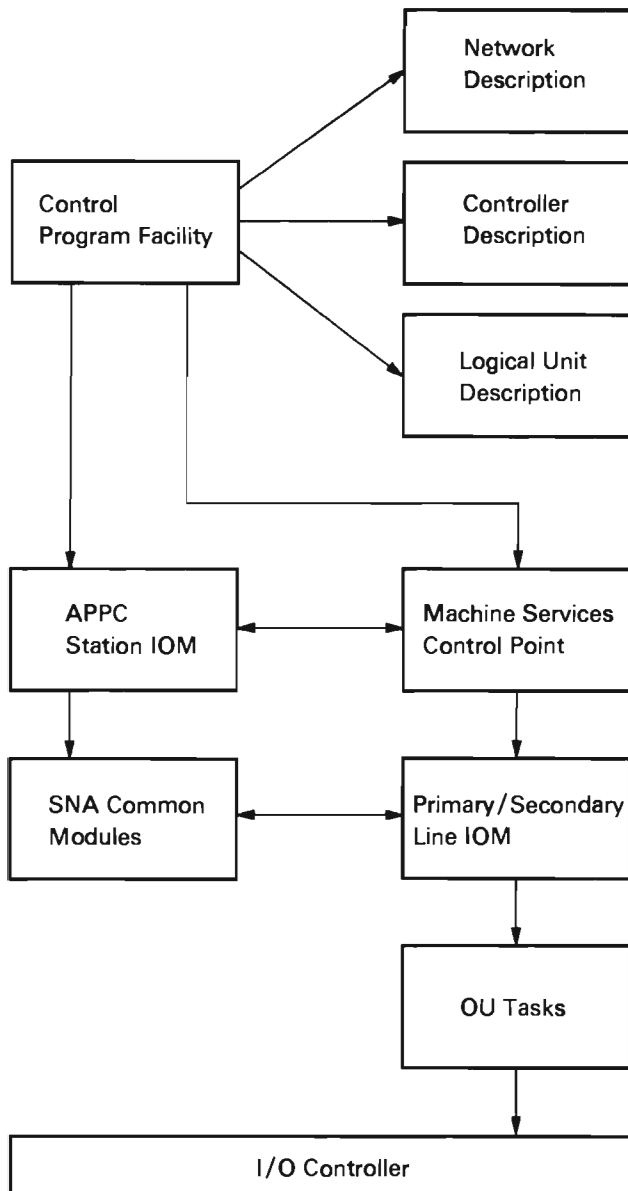


Figure 19-1. APPC Station IOM Overview

The APPC station IOM uses a router module to invoke the appropriate routine. The router gains control when the APPC station IOM task is created (CD vary-on time). The router immediately executes a Receive Message instruction from the APPC station send/receive queue and waits for a send/receive message (SRM). Once an SRM is received, the router performs an if/then/else scan to invoke the routine that processes the requested function. If the function is not identified in the if/then/else scan, the SRM is checked to see if it is a response. If the SRM is a response, the storage occupied by the message is freed. (This message was generated by the APPC station IOM, and no further processing is required.) If the requested function is not supported by the APPC Station IOM, the message is returned with an unsupported status indicated. When the SRM has been processed, the router executes a Receive Message instruction from the APPC station IOM send/receive queue, and the process is repeated.

Because there are unique function codes in the SRMs, the APPC station IOM has one unique callable routine per function and response. A routine is defined for a response only if further processing is required upon the return.

There are three types of APPC station IOM modules:

- System/38 instruction processor modules
- SNA support modules
- I/O support modules

The System/38 instruction processor modules provide direct support of System/38 instructions. These modules are defined as the code that recognizes the requested function and initiates the processing of that function. The SNA support modules are part of the output and input data path. This support is on behalf of the Request I/O instruction. The I/O support consists of modules, including the SNA modules, that are in direct support of the Request I/O instruction. See Figure 19-2 for an overview of the APPC station IOM.

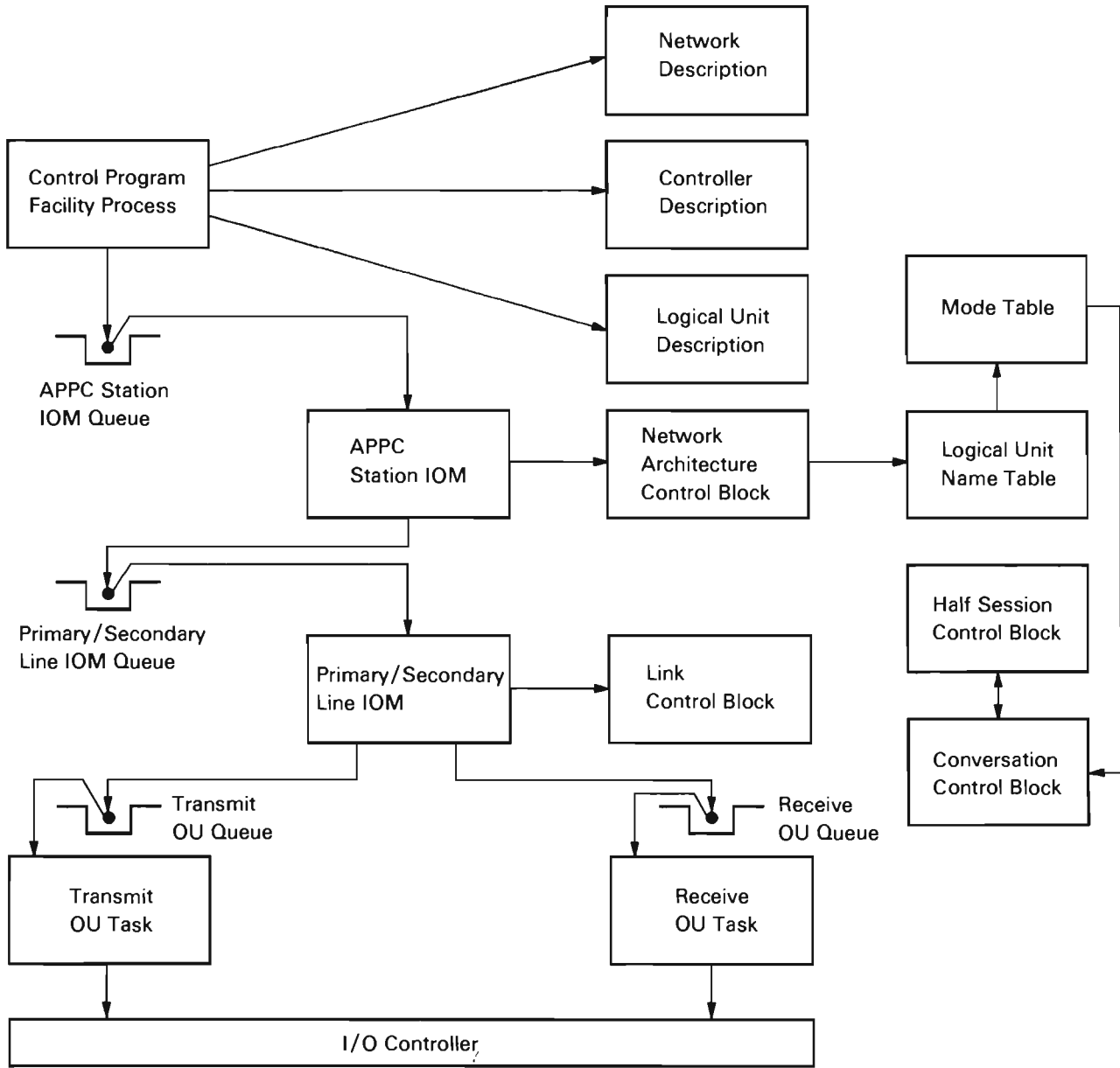


Figure 19-2. APPC Station IOM Processing

System/38 Instruction Support

All source/sink modify instructions passed to the APPC station IOM are in the form of messages (SRMs). These messages are built by mapping the instructions to the VMC format, which is performed by the source/sink instruction processors.

All messages are routed to the appropriate modules by the APPC IOM router. The instructions supported are described in the following paragraphs.

Modify Controller Description (Vary-On/Off): This instruction is used by device management to establish or to break the communication path from the MSCP to the physical unit in the other APPC system.

Modify Controller Description (Dial): This instruction is used to allow a remote Physical Unit to be dialed manually or automatically.

Modify Controller Description (Abandon Connection): This instruction is used to disconnect the APPC station IOM from another APPC system. The APPC station IOM stays active, but the switched line is disconnected.

Modify Controller Description (Continue/Cancel): This instruction is used to allow the reuse or to suspend the reuse of the controller description after an unrecoverable error.

Modify Logical Unit Description (Vary-On/Off): This instruction is used to establish or to break the communications path from the MSCP to the other APPC logical unit.

Modify Logical Unit Description (Activate/De-activate): This instruction is used to de-activate the LU-LU path and is also used to activate the LU-LU path when it is inactive.

The Modify Logical Unit Description (Activate) instruction causes the MSCP to initiate the processing to establish the SNA session resources.

The Modify Logical Unit Description (De-activate) instruction causes all Request I/O instructions except activate resource to be returned to the user return queue with a feedback status indicating the returned condition. The number of request descriptors processed is set in the feedback record. The LU-LU path and LU-LU session are placed in an inactive state with all data being purged for all active conversations. When the necessary synchronization is completed, the conversation identifications are deallocated and the LUD state is set to varied-on state. The active SNA sessions are de-activated.

A Modify Logical Unit Description (activate) instruction is converted to a resume command by the source/sink instruction processor when the LU is inactive (reset).

Modify Logical Unit Description (Reset): This instruction causes each I/O request, except the activate resource request, to be returned to the user return queue with an indicator that the request was not processed. The number of request descriptors (RDs) processed is set in the feedback record, and all available unsolicited data is destroyed. The communications path is placed in an inactive state that can be reactivated with a Modify Logical Unit Description (activate) instruction.

Modify Logical Unit Description (Continue/Cancel): These instructions are used to allow the reuse or to suspend the reuse of the device after an unrecoverable error.

Request I/O: The process that issues conversation requests must communicate with the APPC station IOM by using the Request I/O instruction. The LU-LU session is accessible to the process by using the Request I/O (normal) instruction with the source/sink request object pointer addressing the appropriate LUD and conversation.

The Request I/O instruction is used to perform the normal functions on the LU-LU flow and to manage conversations.

Request I/O (Activate Resource): This request establishes a conversation for the process to communicate with the remote system through an SNA session. The APPC station IOM uses an available source session to request the use of an available remote source session, bids for the use of a target's session, and defines the protocols to start a new source session.

When no session is available, the Request I/O is held pending, provided Normal is the indicated I/O operation, until a session becomes available or a Request I/O (return activate resource) or Request I/O (de-activate resource) is issued to force the Request I/O (activate resource) to be returned.

When a session is allocated to the conversation, the APPC station IOM returns a unique conversation identifier to the process. The process uses the unique conversation identifier for all I/O requests on that conversation to the remote system.

Request I/O (Get Session): This request is used by the process to obtain a new session after the previous session is deallocated from a conversation by a detach request. This request must specify the conversation identifier. The APPC station IOM then allocates a new session to this conversation similar to the way the Request I/O (activate resource) instruction allocates a session to a conversation.

Request I/O (Conversation Reset): This request causes each outstanding I/O request against the conversation to be returned to the process return queue with an indicator that the request was not processed. The number of request descriptors processed is set in the feedback record. The conversation is immediately available for reuse; no Request I/O (activate request) instruction is necessary.

Request I/O (De-activate Resource): This request deallocates a conversation and makes a session (if allocated to the conversation) available to another process. Each I/O request is returned to the process return queue with an indicator that the request was not processed. The number of request descriptors processed is set in the feedback record, and all unsolicited data is destroyed.

Request I/O (Activate Resource Attach Manager): This request allocates a path to the process for the FMH-5 attach requests received from the remote system. With the completion of the Request I/O (activate resource attach manager) instruction, the APPC station IOM returns a unique conversation identifier to the process. The process then issues a Request I/O (receive) instruction using the conversation identifier. When an FMH-5 attach request is received, the APPC station IOM allocates a new conversation, and uses the Request I/O (receive) instruction to return the conversation identifier of the new conversation to the process. The process then uses the new conversation identifier to communicate with the remote system.

Request I/O (Change Number of Sessions): This request causes the total number of sessions, the number of source sessions, and the number of remote source sessions to be increased or decreased.

Request I/O (Return Activate Resources): This request returns to the user's return queue all pending activate resource request I/Os associated with the LUD specified in the source/sink request, with feedback record status indicating the condition.

Request I/O (RECFMS 00 Alert Operations): This request is used to provide SSCP-PU flow alert support when communicating with a 370-type host. Alerts are not supported for connections to another System/38 or APPC peer devices.

Request I/O (Return RECFMS 00 Alert REQIO): This request returns to the user's return queue all RECFMS 00 Alert REQIOs associated with the CD specified in the source/sink request.

Request I/O (Default Operations): This request provides compatibility with current releases of the communications request I/Os. It will be used for APPC functions such as send data, receive, wait on resource, post on receipt, sync check point, prepare to receive and detach.

Request I/O (Request Write): This request sends a signal request to the remote program. This is performed at the machine interface by encoding a request descriptor as a regular or immediate transmit.

Request I/O (Send Response): This request causes the SNA response to be sent. The response sent is either positive or negative based on the setting of the sense data indicator bit in the response header.

Request I/O (Test and Quiesce): This subcommand is used to determine the existence of certain resources and requests to defer the processing of further requests if the specified resources or requests exist at the time the subcommand is processed.

SNA Support

The line IOMs support the data link control layer of SNA. The path control and transmission control layers are supported by the common SNA modules that are invoked by the APPC station IOM. The following features are supported:

- One Physical Unit type 2.1
- Multiple LUs, each with parallel sessions, all of which are LU type 6.2
- Transmission subsystem profile 7
- Function manager profile 19

Path Control

Path control routes basic information units between the remote LU's half-sessions and the System/38 local LU's half-sessions so that the node/link configuration of the network is transparent to the half-sessions. For System/38 inbound data, path control uses information in the format identification transmission header to control delivery of the basic information unit to either the specified supervisory services or a session for an LU in a mode (including a clustered group of LUs).

Path control implements the logical unit description (LUD) as the primary or secondary LU. This view of the LUD as a port uncouples the end-user application from a hard-coded physical address and allows the operating system the option of late coupling of application and physical address. Path control also treats all paths in an LUD in a parallel fashion to provide flexibility in the system's use of the logical links.

For outbound messages, path control constructs a path information unit in the output buffer and sends an output request to the SDLC IOM.

For an input path information unit received from the line IOM, path control ensures that the message unit is large enough to contain the transmission header and request/response header and that the transmission header is a format identification type 2. The local address, session index address assigner indicator, and fields in the transmission header are used to determine the appropriate session (SSCP-PU, SSCP-LU, LU-LU); if the physical unit is active, the path information unit is routed to the connection point manager.

Transmission Control

The transmission control element is a composite protocol that provides control for a locally supported half-session. Transmission control consists of the following SNA components.

Common Session Control: The common session control provides common support for handling flows to half-sessions that are not active. Common session control directs an appropriate activation request to session control for further processing.

Data Flow Control: Data flow control controls the flow of data and function manager data requests and responses between half-sessions. Data flow control handles only data flow control and function manager data requests; network control and session control requests do not flow through data flow control.

The data flow control also uses the request/response header implementation and data flow control state to invoke various resource manager functions.

Resource Manager: This component manages the session and conversation resources. The resource manager initiates the automatic-disconnect function and does preprocessing of the initiate program function.

Connection Point Manager: The connection point manager is the control point within the LU for distribution of request/response units, validating input sequence numbers, maintaining the pacing, and supporting other functions related to the half-session flows.

For outbound messages, the session and data traffic states are checked. Additional checking is performed before the message is sent to path control:

- On the expedited flow, the connection point manager forwards a request only if no response to a previously expedited request is due from the remote system.
- On the normal flow, the size of the request/response unit is validated.
- When secondary-to-primary pacing is supported for a normal flow request, the connection point manager determines when to set the pacing indicator in the request header to indicate that a pacing response must be returned. The connection point manager then prevents the forwarding of additional outbound normal requests until the pacing response is received.
- When primary-to-secondary pacing is supported, and if a pacing request has been received, the connection point manager sets the pacing indicator on in the response header in a normal flow response to notify another APPC system that additional normal flow requests may be sent.

When session and data traffic states are not active, the connection point manager forwards the permitted inbound SNA activation requests (BIND) to common session control. For other inbound messages, the connection point manager checks the request/response header to determine if pacing or expedited requests can be sent. When a pacing response should be sent to another APPC system, the size of the request/response header is validated, the sequence number in the transmission header is compared with the expected value, and a flag is set to indicate that a pacing response should be sent.

Session Control: Session control supports protocols related to session and data traffic activation, de-activation, and recovery. All session control requests flow from the primary unit to the secondary unit, and all responses flow from the secondary unit to the primary unit.

For an outbound response, session control identifies the response code and sets the appropriate indicators according to whether the response is positive or negative. The response is forwarded to path control.

The APPC station IOM supports the following SNA session control requests:

- BIND to establish a LU-LU session.
- UNBIND to break a LU-LU session.

Requests pertaining to LU-LU sessions are sent to the end user via I/O requests for additional processing.

The APPC communications to a physical unit Type 4/5 performs the following for SNA session control requests from the host.

- Activates a physical unit to establish an SSCP-physical unit session.
- De-activates a physical unit to break an SSCP-physical unit session.
- Activates a logical unit to establish an SSCP-LU session.
- De-activates a logical unit to break an SSCP-LU session.

I/O Support

Output

The output process is started by a request I/O queueing function (#T2RQIO). This function receives all I/O requests from the APPC station IOM queue and enqueues the request on the correct conversation queue. The requests are always enqueued last on this queue and are categorized according to the I/O path on which they are to be executed (for example, expedited or normal transmit, expedited or normal receive, receive-any). The queuing function uses the second byte of the message key to categorize the requests.

Once the requests are enqueued, the scheduler (#T2SCED) is invoked to process the output. The scheduler consists of two parts: loop selection and logical I/O path selection. Loop selection uses the conversation control block as its basic unit. This module multiplexes output for different SNA sessions into one output request and ensures that each conversation has equal opportunity for output.

The logical I/O path selection routine (#T2BSTO) gains control from the loop selection module when a conversation control block has I/O requests to be processed. This module locates the next request to be processed (for example logical I/O path), enforcing the path control rule that expedited processing is executed before normal processing. The selection routine also controls the building of the SNA frames.

The transmit path message pointers in the half-session control block are examined and if the pointer is 0, the appropriate 2-byte key is built and the conversation queue is searched to find an I/O request. When an I/O request is located, the output SNA frames and associated buffer control lists are built, and control is returned to the loop scheduler.

On return, the loop scheduler determines if it is time to build an output request. If not, the loop scheduler selects the next conversation control block and repeats the selection sequence. If the output request is to be built, the loop scheduler builds the request, puts the buffer in a busy status, and sends the output request SRM to the line IOM.

Output Posting

The APPC station IOM now waits for the output to complete or for the input to arrive; output requests have a higher priority than input requests. A transmit/receive Request I/O instruction is first encoded as a transmit Request I/O instruction; then, when all transmits are complete, the instruction is encoded as a receive-only Request I/O instruction and enqueued first on the conversation queue. The output request/response process uses the half-session control block pointer to locate the Request I/O instruction associated with the frame. Then, using the indexes in that entry, it locates the request descriptor (RD) associated with the SNA frame and marks it processed.

At this point the Request I/O instruction can be in three states:

- More transmit RDs to process
- All transmits complete, but receives yet to process
- All transmits and receives complete

The appropriate action is then taken by the APPC station IOM to complete the output request/response process. At this time, the buffer is marked not busy.

On return, the loop scheduler is invoked if more activity is scheduled.

Input

The APPC station IOM does not explicitly request input from the transmission line. The line IOM responds to polls from the primary unit and, in effect, the APPC station IOM has a read operation outstanding. The line IOM passes received information frames to the APPC station IOM via an input message. This message contains the location and number of valid information frames stored in the input buffer. The input routine processes the information frames one frame at a time until all frames are processed. The input routine uses the transmission header to locate the correct half-session control block and then determines the logical I/O path on which the frame is to be sent. If an I/O request is not pending or if no buffer space is available, the frame is considered unsolicited data, and an event or a feedback record is returned to the user. The user is informed of only the first frame of unsolicited data; however, subsequent frames can cause the same situation, depending on whether the I/O request contained enough buffer space to contain the frames.

Unsolicited data is held on the queue of the conversation control block in the form of a message. This message is built by the APPC station IOM and has the same format as an input message. The message also contains a data area for the frame, allowing the APPC station IOM to free the input buffer of the line IOM even if a user buffer for the data does not exist. Unsolicited data is processed as an input request by the same modules that perform input processing. The conversation control block contains indicators to inform the loop scheduler to process the data.

Error Logging

The APPC station IOM does not keep statistics concerning the station; statistics are kept by the line IOM. The APPC station IOM does log SNA path errors. These errors are recorded using the format of the error recording functions. The data contains portions of the SNA frame, including the transmission and request headers, the sense data, and the first 14 bytes of the request/response unit.

DATA AREAS

Refer to *Source/Sink Data Areas* in the *VMC Overview* section of this manual for descriptions of source/sink data areas. Also, refer to *Data Areas* in the *Instruction Processors* section of this manual for descriptions of the ND, CD, and LUD.

Network Architecture Control Block

The network architecture control block is a common control area used to manage the systems network architecture (SNA) portions of the APPC station. It contains a subset of data in the CD and is always in main storage when the APPC station task is executing. The network architecture control block functions as a directory to areas built in machine-wide storage. It also functions as a collection point for vital converged station characteristics, a control point for converged station output, and a common location for unique SNA and converged station work areas.

The network architecture control block is allocated from machine-wide storage at vary-on CD time.

Logical Unit Name Table

The logical unit name table is used as a collector for all information necessary to operate the logical unit. This entry represents the LU such that the local LU and the remote LU can communicate through logical groupings of conversations.

The logical unit name table is accessed externally through the system pointer to the LUD and accessed internally through the LUD I/O index contained in the LUD. The LUD I/O index is set in the LUD and in the conversation identifier at vary-on LUD time. The LUD I/O index is a direct index into the logical unit name table. The first logical unit name table entry represents the physical unit. It is accessed with an index of 0.

Each logical unit name table entry contains the logical unit type 6.2 for purposes of routing to the proper SNA modules and building the conversation identifier for LU 6.2.

The logical unit name table serves as an anchor for groupings of SNA sessions and the conversations used to access the SNA sessions. All contention winner primary half-sessions have a preassigned local address which is the destination address field in the LUD. The storage for the mode table and conversations is allocated in machine-wide storage. This is done at vary-on LUD time for LU type 6.2.

The logical unit name table entries representing the LUDs have one mode entry for each mode specified in the LUD device-specific area. It also has a group of conversations that consists of the total of the user conversations specified for each mode, one attach manager conversation for each mode and one conversation for the SSCP-LU session. The SSCP-LU conversation has a conversation index of 0. The attach manager conversations have indexes equal to the mode index they are associated with.

Mode Table

The mode table is used to manage a group of conversations and sessions that share a common set of operational and functional characteristics such as, pacing limits and maximum length of RU on LU-LU sessions. Each mode table entry is associated with a definite LU through the backward pointer of the logical unit name table entry.

The mode entry serves as the major work area for the conversation manager. The mode table contains a copy of the bind image used to bind the primary half-sessions and a copy of the negotiated bind response. The negotiated bind response is used to communicate the bind fields (security/access) to the target conversations and to ensure the same characteristics for all sessions existing under this mode entry.

The conversations through which an application accesses SNA sessions are associated with a particular mode entry. These conversations are managed by the conversation group pointer, conversation free list, and change-number-of-sessions data. The conversation group pointer serves as an access to the conversations assigned to this mode entry. These conversations are contiguous in machine-wide storage. They are a subset of all conversations associated with this logical unit name table. The conversation free list contains the conversations that are deallocated. This list is initialized at vary-on LUD time to all possible conversations for this mode. When a conversation is allocated, the first conversation on the list is used to build the conversation identifier. When a conversation is deallocated, the conversation is placed back in the conversation free list. The change-number-of-sessions data is used to manage the session and conversation allocation/deallocation.

Half-Session Control Block

Half-session control blocks are used for routing transmit and receive requests to the proper network addressable unit and for routing input to the proper destination in the System/38. Each half-session control block represents an SNA session; supports the SNA transmission subsystem; and contains fields that support path control (routing and expedited/normal SNA paths), connection point manager (pacing), and LU (I/O queuing and session states).

Half-session control blocks are allocated and initialized in machine-wide storage at vary-on LUD time for logical unit type 6.2 LUDs.

Each half-session control block represents an SNA path for data transmission. The half-session control blocks are attached to a conversation by a Request I/O (activate resource) instruction or detached by a Request I/O (deactivate resource) instruction. The session may be bound for a certain mode, in which case the half-session control blocks are placed on a free list in the mode table; otherwise, they are either on the half-session control block available list or assigned to a conversation.

The first two half-session control blocks are reserved for boundary function support and internal APPC IOM usage. The first supports an SSCP-physical unit session that is activated and ready for I/O traffic at CD vary-on time. The second half-session control block is used as a temporary output holding area when the output structure is already busy processing an output request.

Conversation Control Block

The conversation control block represents the conversation resource viewed through a conversation identifier. The maximum number of conversations is specified as an attribute of the mode entry in the LU type 6.2 LUD.

The conversation identifier is used to access the conversation for all machine-interface operations, and the backward pointer in the half-session control block is used to access the conversation for all input operations. When the conversation control block is not connected to a half-session, it is an asynchronous conversation. When a conversation is asynchronous, the half-session control block pointer is 0.

Conversation Identifier

The conversation identifier is used as the access to the conversation and its resources. The conversation identifier is located in the source/sink request variable data area and is addressed through the Request I/O instruction.

STRUCTURE

The following is a list of the APPC station IOM modules and the function that each module performs. The list also shows how the module is invoked.

#NA2BRP Build And Send BIND Response

Function: Builds and sends positive or negative BIND response depending on sense code passed from LU resource manager. (This procedure is called for LU type 6.2 only.)

How Invoked: Within this component.

#NA2CPLU Activate/De-activate LU Response Processor

Function: Updates LU state for activate and de-activate LU requests.

How Invoked: Within this component.

#NA2CPMR Connection Point Manager Receiver

Function: Performs connection point management checks (including pacing) for data received and routes for further processing.

How Invoked: Within this component.

#NA2CPMS Connection Point Manager Sender

Function: Performs connection point management checks for data to be transmitted.

How Invoked: Within this component.

#NA2CPPU Activate/De-activate PU Response Processor

Function: Updates physical unit state for activate physical unit and de-activate physical unit requests.

How Invoked: Within this component.

#NA2DFCR Data Flow Control Receive

Function: Ensures the data flow control protocol enforcement of the LU type 6.2 receive.

How Invoked: Within this component.

#NA2DFCS Data Flow Control Send

Function: Ensures the data flow control protocol enforcement of the LU type 6.2 send.

How Invoked: Within this component.

#NA2EROR SNA Error Processor

Function: Determines if negative response must be returned, and if so, builds negative response unit and routes for transmission.

How Invoked: Within this component.

#NA2PCR Path Control Receiver

Function: Validates transmission header data and routes path information unit received to connection point manager receive.

How Invoked: Within this component.

#NA2PCSD Path Control Sender

Function: Builds path information unit in output buffer and indicates output is pending.

How Invoked: Within this component.

#NA2RFMS Request/Record Formatted Maintenance Statistics

Function: Responds to REQMS request received, if necessary, then builds record formatted maintenance statistics request as a pseudo-I/O request for transmission.

How Invoked: Within this component.

#NA2SCSD Session Control Sender

Function: Identifies session control response, updates appropriate [finite state machine] states, and forwards response to path control send.

How Invoked: Within this component.

#TP2SECS Station I/O Manager Router

Function: This module interrogates all incoming send/receive messages and routes each to the appropriate module.

How Invoked: Within this component.

#T2AHDR Attach Header Received

Function: Processes the attach header received on the SNA session.

How Invoked: Within this component.

#T2ALUR Activate LU Response Processor

Function: Interrogates Activate LU-received response from MSCP to-establish SSCP-LU and LU-LU half-session controls and initializes transmission and response headers for SNA activate LU response.

How Invoked: Within this component.

#T2APUR Activate PU Response Processor

Function: Interrogates activate physical unit-received response from MSCP to establish output structure storage and initializes transmission and response headers for SNA activate physical unit response.

How Invoked: Within this component.

#T2ASRM Asynchronous Resource Manager Request
I/O Processor

Function: Processes activate resource request, get session request I/Os from input queue and mode entry's waiting queue, routes change-number-of-sessions, and test and quiesce request I/Os to #T2CNOS.

How Invoked: Within this component.

#T2AVR2 Activate Resource Send/Receive Message
Processor

Function: Processes response to BIND received.

How Invoked: Within this component.

#T2BDME BIND Determine Mode Entry

Function: Finds mode table entry associated with mode name specified in the received BIND.

How Invoked: Within this component.

#T2BDRI Build Dummy I/O Request

Function: Creates a pseudo-I/O request structure and enqueues it to the dummy conversation control block for transmission.

How Invoked: Within this component.

#T2BIDRP BID Response Received Processor

Function: Processes the BID response received on a session.

How Invoked: Within this component.

#T2BIDRQ BID Request Received Processor

Function: Processes BID request received on a session.

How Invoked: Within this component.

#T2BISRQ BIS Request Received Processor

Function: Processes the BIS request received on a session.

How Invoked: Within this component.

#T2BNDRQ BIND Request Received Processor

Function: Process the BIND request received for a session.

How Invoked: Within this component.

#T2BND0 BIND Request Scheduler

Function: Schedules the BIND request that is to be sent to a session.

How Invoked: Within this component.

#T2CMDS Route Secondary Station Commands

Function: Interrogates all non-I/O SRMs and response SRMs and routes each to the appropriate routine. Contains the following internal routines:

- #T2ACTS: Processes activate session SRM.
- #T2DACS: Processes de-activate session SRM.
- #T2VOFL: Processes vary-off LUD SRM.
- #T2VOFC: Processes vary-off CD SRM.
- #T2VONL: Processes vary-on LUD SRM.
- #T2CONT: Processes Request I/O (continue) SRM.
- #T2QUISC: Processes quiesce SRM.
- #T2RSET: Processes reset SRM.
- #T2RSUM: Processes resume SRM.
- #T2SRTO: Processes send/receive timeout SRM.
- #T2SPND: Processes suspend SRM.

How Invoked: Within this component.

#T2CNOS Change Number Of Sessions Processor

Function: Decodes the action field in the change-number-of-sessions entry in the source/sink description and performs the requested function.

How Invoked: Within this component.

#T2DFCF Data Flow Control Flag Interface Processor

Function: Processes the flags set by #NA2DFCR to act either as an interface to the conversation manager or to process delayed data.

How Invoked: Within this component.

#T2DIS Process SNA Clean-up for SDLC Disconnect Command Received

Function: Sends abnormal disconnect SRM to MSCP for normal and abnormal disconnect SRMs. Sends abnormal de-activate physical unit SRM to MSCP for the second Set Normal Response Mode instruction.

How Invoked: Within this component.

#T2DLYD Process Delayed Data

Function: Processes data delayed due to between-bracket conditions.

How Invoked: Within this component.

#T2DOWN Report Station Failure Condition

Function: Marks a station failure condition and reports it via a feedback record if a Request I/O instruction is outstanding.

How Invoked: Within this component.

#T2DQBND Dequeue BIND Requests

Function: Dequeues and processes BIND requests on the LU BIND queue for the associated mode name.

How Invoked: Within this component.

#T2DRSP Data Flow Control Response Sender

Function: Builds a response for output based on data in the network address control block.

How Invoked: Within this component.

#T2DSAMT Materialize LUD Device-Specific Area

Function: Builds the materializable device-specific area in the LUD.

How Invoked: Within this component.

#T2DSAV Verify LUD Device-Specific Area

Function: Verifies the user template for the LUD device-specific area.

How Invoked: Within this component.

#T2ERLG Build Error Log Entry

Function: Builds an error log entry SRM and sends it to be logged.

How Invoked: Within this component.

#T2ERPR Error Processor

Function: Builds and sends the UNBIND request into the network architecture control block and sends it to #NA2SCSD.

How Invoked: Within this component.

#T2FDBK Feedback I/O Request Result

Function: Builds the feedback record for an I/O request feedback SRM.

How Invoked: Within this component.

#T2FRSES Free Session Received

Function: Processes a session-between-brackets condition.

How Invoked: Within this component.

#T2IOS Process Inoperative State Request Received from Secondary Line IOM

Function: Sets station offline and calls #T2DOWN to notify the machine interface of a station failure.

How Invoked: Within this component.

#T2IPIU Input Path Information Unit Processor

Function: Routes received SNA path information unit for validation, then moves data into receiving I/O request buffer area.

How Invoked: Within this component.

#T2OUTR Output Request Response Processor

Function: Matches output request response to I/O Request and sends feedback record when processing is complete.

How Invoked: Within this component.

#T2OUTX Forward Output Request to Secondary Line IOM

Function: Completes and sends output request SRM to the secondary line IOM.

How Invoked: Within this component.

#T2PBAR PREBIND Activate Resource

Function: Performs resource manager function for the modify LUD activate session and modify LUD continue requests.

How Invoked: Within this component.

#T2PGAR Purge Activate Resource Request I/Os

Function: Returns all activate resource request I/Os.

How Invoked: Within this component.

#T2PGCB Purge Logical Unit Activity

Function: Sends feedback record with partially complete or complete status for I/O request during Modify LUD (reset) instruction processing.

How Invoked: Within this component.

#T2RQIO I/O Request Processor

Function: Enqueues an I/O request with the correct flow key to the proper half-session control block queue.

How Invoked: Within this component.

#T2RTFN Request Information Unit Router

Function: Examines the current function and function stage fields and routes control to the proper response module. This module also loads the request I/O flow field pointer and passes control to the proper SNA module.

How Invoked: Within this component.

#T2RTRRP Ready-To-Receive-Response Received

Function: Processes the ready-to-receive-response received on a session.

How Invoked: Within this component.

#T2RTRRQ Ready-To-Receive-Request Received

Function: Process the ready-to-receive-request received on a session.

How Invoked: Within this component.

#T2SCED Activity Scheduler

Function: Removes the conversation control blocks from the output scheduling queue and dispatches transmit I/O requests.

How Invoked: Within this component.

#T2SEH APPC Station IOM Exception Handler

Function: Identifies the exceptions encountered in the secondary station and returns them to the mainline RECM in #TP2SECS when the exception is caused by the user.

How Invoked: As fourth level exception handler.

#T2SSDEH APPC Station Source/Sink Data Exception Handler

Function: Handles any source/sink data exceptions that occur in the APPC station IOM.

How Invoked: From the third level exception handler.

#T2SYRI Synchronous Request I/O Send/Receive Message Processor

Function: Processes all synchronous request I/O send/receive messages.

How Invoked: Within this component.

#T2UNBRQ UNBIND Request Received Or Sent

Function: Processes the end of a session (UNBIND request sent or received) with respect to the resource manager function.

How Invoked: Within this component.




Binary Synchronous Communications I/O Manager

INTRODUCTION

The binary synchronous communications (BSC) I/O manager (IOM) activates, manages, and de-activates the BSC telecommunications link and enforces BSC protocol. One BSC IOM task exists for each BSC telecommunications link.

The BSC IOM interfaces with the following:

- The machine services control point (MSCP)
 - The error log
 - An I/O controller
 - Diagnostic component
 - Modify Network Description instruction
 - Modify Controller Description instruction
 - Modify Logical Unit Description instruction
 - Request I/O instruction
- 

A BSC IOM task is created by the MSCP as a result of a Modify Network Description (vary on) instruction. The BSC IOM task is associated with one communications I/O controller (IOC) line position and is shown in Figure 20-1.

The BSC IOM is used to communicate with devices on a switched point-to-point line, a nonswitched point-to-point line, and a multipoint line as a tributary station. Up to 32 sessions per line can be supported on tributary.

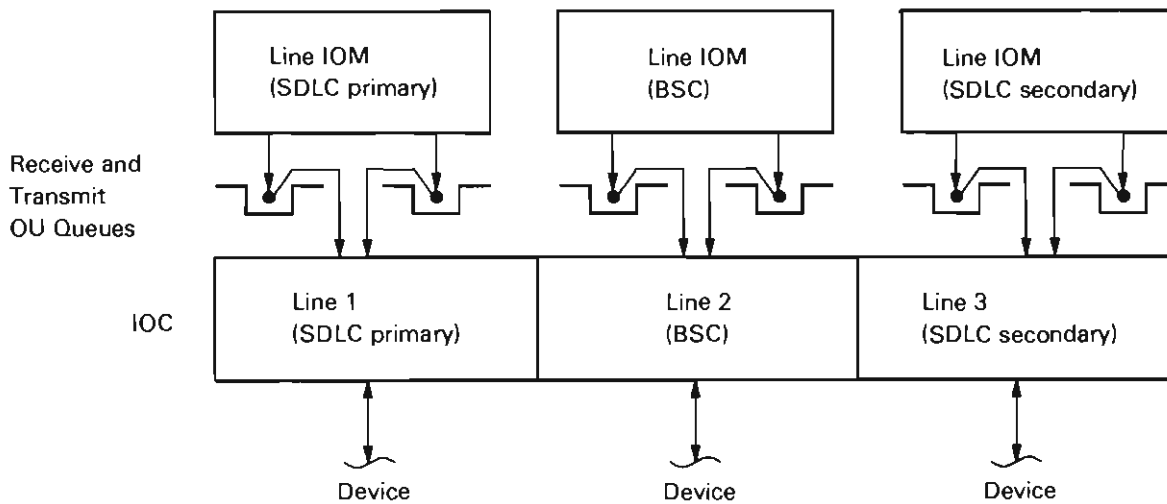


Figure 20-1. BSC IOM/IOC Line Position Relationships

Communication with external components is through a send/receive message which the BSC IOM receives through a single send/receive queue as shown in Figure 20-2. The message can be generated in three ways; by an external VMC or diagnostic component, an operational request element (ORE), or a Request I/O instruction.

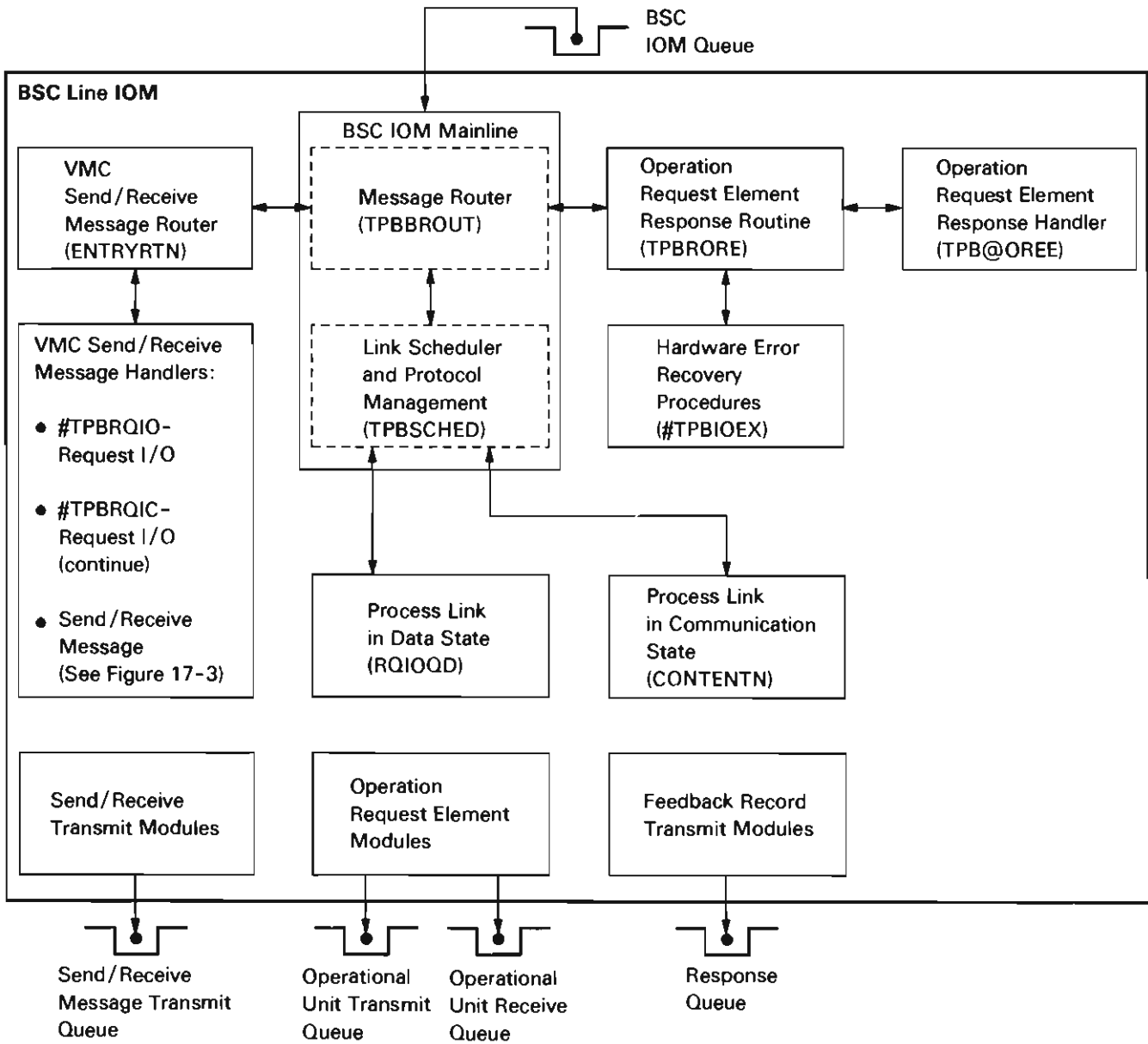


Figure 20-2. BSC IOM Internal Structure

The message router receives the message and uses the key field in the message to determine if the message is generated from an external VMC or diagnostic component, an ORE, or a Request I/O instruction. Then, based on the function field of the message, the message router invokes the appropriate message handler for the messages generated by the Request I/O instruction and the external VMC or diagnostic components.

The messages generated by the Request I/O instruction are routed to routine #TPBRQIO in module #TPBBIOM. There the messages are placed on an internal queue for later execution. Request I/O (continue) instruction messages are routed to routine #TPBRQIC in module #TPBBIOM. The messages generated by an external VMC or diagnostic component are routed to a message specific routine for processing. See Figure 20-3 for a list of send/receive message (SRM) handling entry points.

Function	Routine (Entry Point)
Timer response	#TPBTIME
Activate session	#TPBASES
Reqio continue	#TPBRQIC
De-activate session	#TPBDSES
Reset session	#TPBRSES
Vary off LUD	#TPBVOFL
Discontact	#TPBDCON
De-activate link	#TPBVOFN
Initialize line	#TPBENB
De-activate connect in	#TPBDABL
Connect in	#TPBESC
Vary on LUD	#TPBVONL
Contact	#TPBCON
Exchange identification	#TPBXID
Connect out	#TPBDIAL
Abandon connection	#TPBABCN
Abandon connect out	#TPBABO
Change network description retry sets	#TPBNDRT
Activate link	#TPBVONN
Machine interface timer message	#TPBSOFT
Modify device specific	#TPBMDSA
Modify unit specific	#TPBMUSC
Modify line specific	#TPBMLSC
Resume session	#TPBRSUM
Quiesce session	#TPBRSES
Suspend session	#TPBSPND
On-line test request	#TPBOLTA
Read data store	#TPBRDSO
Internal trap	#TPBTRPO
Diagnostic control	#TPBDI
Cancel Invite	#TPBSINV
Return message (illegal entry)	#TPBBAOM

Figure 20-3. Send/Receive Message Handling Entry Points

The messages generated by the OREs are handled differently. The queue message router relies on the routine that generated the ORE to provide the address of the routine to process the ORE response. See Figure 20-4 for a list of ORE response handling entry points.

Function	Routine (Entry Point)
Perform BSC protocol analysis	#TPBTXTR
Process response from sending end of transmission	#TPBABRR
Process ORE response for unsolicited line bid	#TPBUNSL
Process OREs for sending end of transmission	#TPBABRT
Perform BSC multi-point tributary analysis	#TPBOPMP
Process ORE response for reset	#TPBBID1
Process set line priority/reset/initialize ORE response	#TPBVNN1
Finish vary off ND, destroy IOM task	#TPBVOF1
Initialize I/O controller ORE response	#TPBENB1
Process enable switch connection response	#TPBESC1
Process identification received	#TPBXID1
Finish de-activate connect in	#TPBDAB1
Contact (identification exchange) processing	#TPBCON1
Contact (identification exchange) processing	#TPBCON2
Process dial ORE response	#TPBDL1
Finish abandon connect out processing	#TPBAB01
Finish abandon connection	#TPBABC1
Finish vary off LUD	#TPBVFL1
Online test responder bid for line	#TPBOLT1

Figure 20-4 (Part 1 of 2). ORE Response Handling Entry Points

Function	Routine (Entry Point)
Online test responder bid response	#TPBOLT2
Online test requester bid response	#TPBOLTD
Online test requester test request response	#TPBOLTE
Online test requester receive line bid	#TPBOLTF
Online test requester line bid setup	#TPBOLTG
Online test end of transmission response before cleanup	#TPBOLTH
Online test responds to text message received	#TPBOLTR
Online test sends the test message	#TPBOLTS
Read data store ORE response handler	#TPBRDS2 ¹
Internal trap ORE response handler	#TPBTRP2 ¹
Write poll list response handler	#TPBWPL1 ¹
Reset write poll list response handler	#TPBWPL2
Diagnostic dial	#TPBWERA
Diagnostic identification after dial	#TPBEARS
Diagnostic end of transmission after dial	#TPBE07
Diagnostic enable switch connection	#TPBECAA
Diagnostic identification after answer	#TPBWCAN
Diagnostic read	#TPBDRFT
Diagnostic read line bid	#TPBDRUN
¹ OREs routed by direct call, not TPB@OREE value.	

Figure 20-4 (Part 2 of 2). ORE Response Handling Entry Points

The link scheduler and protocol management routines are invoked by the message router when the message has been processed. The link scheduler and protocol management routines build the OREs for transmitting data or responses and sends the OREs to the operational unit queues.

All BSC IOM and IOC detected errors are processed by the error recovery routines.

DATA AREAS

Link Control Block

The link control block (BLKB) is the primary control block for the BSC IOM. It is allocated in machine-wide storage when the BSC IOM task is created and exists until the task is destroyed. The BLKB contains the following data and control areas:

- Feedback record parameter area
- Pointers to other objects and control areas
- Status flags and counters
- Link control characters (EBCDIC or ASCII)
- Work areas for the various BSC IOM routines
- Operation request elements (ORE)
- Program operation blocks (POB)
- Function operation blocks (FOB)
- Message operation blocks (MOB)
- Error and timer messages

Service Order Table

The service order table (BSOT) is the secondary control block for the BSC IOM. It is allocated in machine-wide storage at vary-on LUD time. One SOT exists for each LUD that is varied on. The BSOT contains information related to one session such as the request I/O hold queue used during active sessions, a copy of pertinent attributes of the device from the logical unit description, and the logical unit description session status. For point-to-point connections, one LUD exists. For multipoint connections, up to 32 LUDs can exist.

Controller Description Table

The controller description table is a control block containing information related to the station controller description. It contains a pointer to the controller description, a pointer to the statistical data record, and various status fields. For each controller description that is varied on, there exists one controller description table.

Operation Request Element

The BSC IOM communicates with the IOC by way of a send/receive message called an ORE. In the operation block portion of the ORE the various commands are specified, data areas are indicated, and status is returned. Three types of operation blocks are used: The function operation block, the program operation block, and the message operation block.

The function operation block contains single commands such as Initialize, Establish Switched Connection, and Line Reset, as well as Write and Read commands.

The program operation block is used when multiple function operation blocks are to be executed. The program operation block references a chain of function operation blocks, each of which contains a command to be executed.

The message operation block is used during data transfer to eliminate the chance of command time-outs when two separate commands must be issued to the IOC for the execution of one I/O operation.

STRUCTURE

The following is a list of the modules in the BSC IOM and the function that each module performs. This list also shows how the module is invoked.

#TPBBIOM BSC IOM Mainline Routines

Function: Activates, manages, and de-activates a BSC telecommunication link. Performs all protocol management and link level recovery. Handles the scheduling and routing of I/O requests.

How Invoked: Other VMC components.

#TPBDE BSC Online Test

Function: Provides diagnostic routines for BSC online test request response read data store, and trap functions. Allows the data link to be tested in a manner transparent to the user application program.

How Invoked: Within this component.

#TPBDI BSC Diagnostic Mode Control

Function: Provides control for line operation in diagnostic mode. Allows link to perform online test, trap, and read data store operations without an application program to drive the link.

How Invoked: Within this component.

#TPBELSE BSC Nonmainline Paths

Function: Performs auxiliary functions for the main BSC IOM module. These include routines to handle timer requests, ASCII translation, and line abort, and sends all messages to the MSCP.

How Invoked: Within this component.

#TPBERPL BSC Error Recovery Procedures Processor

Function: Handles errors on the BSC link resulting from horizontal microcode detected errors. Initiates recovery procedures for I/O errors, OU errors, channel errors and invalid commands. Updates the retry counters and logs the error when the retry count has been exceeded.

How Invoked: Within this component.

#TPBLPER BSC Link Protocol Error Recovery Procedures Processor

Function: Examines the contents of the receive buffer and initiates a recovery action based on what control sequence or text transmission was received. Updates the retry counters and logs the error when the retry count has been exceeded.

How Invoked: Within this component.

#TPBMODC BSC Modify Controller Description Processor

Function: Processes messages sent to the main BSC IOM module as a result of a modify CD request. Establishes contact with the remote station and handles the abandon connection at the end of a session. Also performs the dial operation for a switched connection.

How Invoked: Within this component.

#TPBMODL BSC Modify Logical Unit Description Processor

Function: Processes messages sent to the main BSC IOM module as a result of a modify LUD request. Handles the vary on/vary off of the LUD and assumes responsibility for the activation, de-activation, and resetting of a session.

How Invoked: Within this component.

#TPBMODN BSC Modify Network Description
Processor

Function: Processes messages sent to the main BSC IOM module as a result of a modify ND request. Handles the vary on/vary off of the ND and establishes the connection.

How Invoked: Within this component.

#TPBMTPT BSC Multipoint Function Processor

Function: Handles poll/select responses and performs poll list management

How Invoked: Within this component.

Channel I/O Manager

INTRODUCTION

The channel I/O manager (IOM) is a component that issues channel commands, logs channel hardware errors and event handler errors, notifies IOMs of post-event attention request, and (for devices other than auxiliary storage) participates in I/O error recovery. The channel IOM has two primary functions:

- Start/halt device function that issues start and halt device channel commands, handles errors that occur as a result of these commands, and informs the IOMs of the success or failure of the operation.
- Channel event processing function that issues read event and start channel commands, logs channel hardware and event handler errors, notifies IOMs of post-event-attention and error requests, and operational unit (OU) task failures, and (for devices other than auxiliary storage) participates in I/O error recovery.

Start/Halt Device Function

The start/halt device function supports the start and halt device commands issued by a device IOM. This support includes related processing and error recovery.

As shown in Figure 21-1, the start/halt device function consists of an IOM task, OU task, IOM queue, and an OU queue. The address of the IOM queue (IOSRQCSH) is in the machine communications area (field MCA4RSHO). The address of the OU queue (OUSRQCSH) is also in the machine communications area (field MCA4RSHI). Both queues, plus the OU task dispatching element (TDE), are in #RTTASKS. The OU task base registers are stored in the TDE. The IOM task used is the device IOM task of the device issuing the start or halt device request. Because no events (except error events for the channel event processing function) are posted as a result of a start or halt device channel command, the start/halt device function shares the channel error function queue control table (QCT).

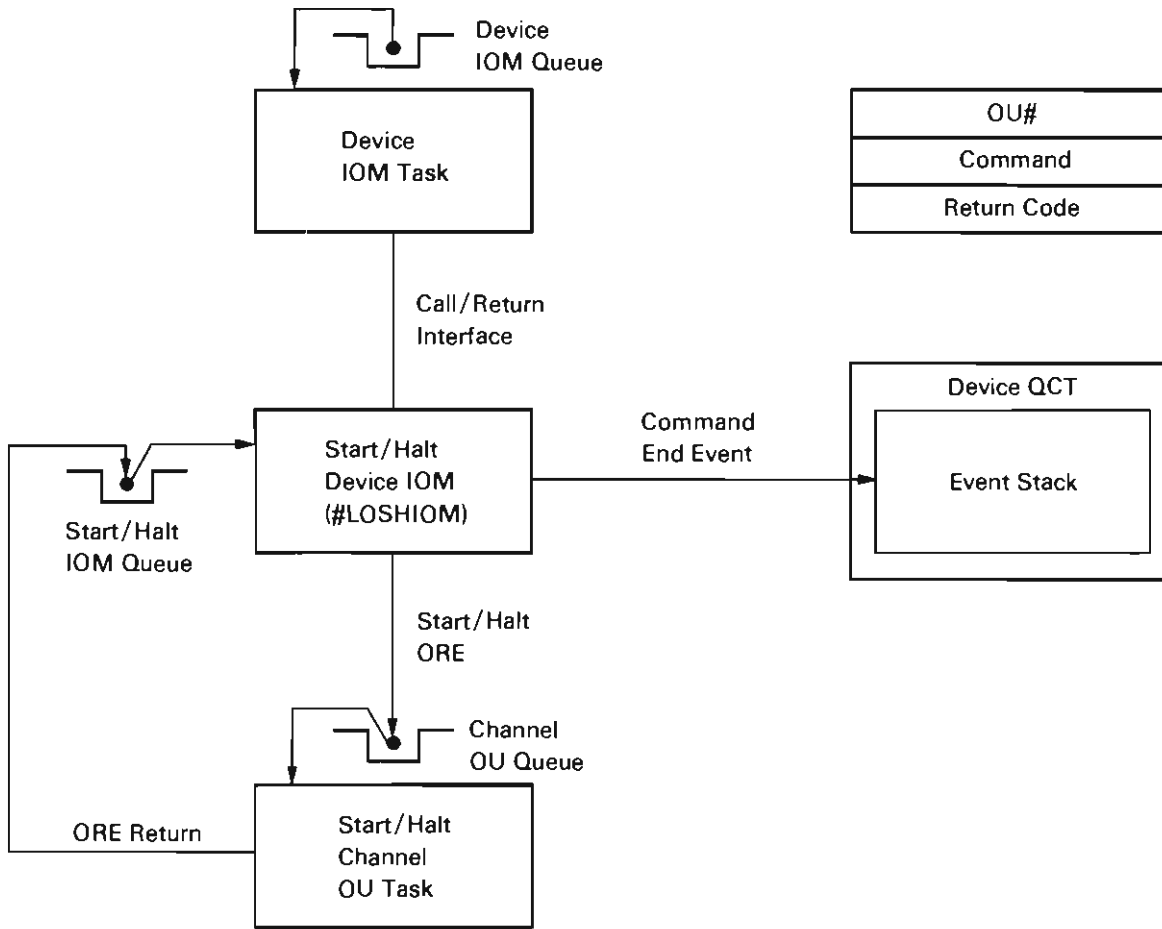


Figure 21-1. Start/Halt Device Function

The start/halt device function is initiated when a device IOM invokes module #LOSHIOM.

#LOSHIOM validates the input, and then builds and sends a start/halt device operation request element/function operation block (ORE/FOB) to the OU queue. When the ORE/FOB is returned, #LOSHIOM checks bytes 14 and 15 of the ORE for one of the following indications:

- 4007

Meaning: The previous start/halt command did not complete.

Action: #LOSHIOM retries the command. After four unsuccessful retries, the operation is considered uncorrectable; the return code is set to hex 02 and control returns to the invoking IOM.

- 4010

Meaning: The start/halt request has timed out, possibly due to a nonfunctioning channel.

Action: #LOSHIOM reinserts the OU number into the ORE/FOB and retries the operation once. If another time-out occurs for this request, the operation is considered uncorrectable; the return code is set to hex 04 and control returns to the invoking IOM.

- 40xx (where xx is neither 07 nor 10)

Meaning: Other operational program error.

Action: #LOSHIOM rebuilds the ORE/FOB and retries the operation once. If another 40xx is returned on this request, the operation is considered uncorrectable; the return code is set to hex 08 and control returns to the invoking IOM.

- 0100

Meaning: Command has been accepted and executed (command end).

Action: #LOSHIOM determines if this was a halt device command. If so, #LOSHIOM determines if an OU task is waiting on the send/receive counter in the QCT, and if so, #LOSHIOM posts a command end event to the QCT event stack.

Before returning control to the caller, #LOSHIOM sets flags in the return code area to indicate the status of the operation. If an invalid OU number has been passed to #LOSHIOM, the return code is set to hex 80 to indicate invalid input data.

Channel Event Processing Function

Figure 21-2 shows an overview of the channel event processing function. This function consists of three TDEs, three queues, and a QCT. The TDEs are for the IOM task (called the resident-channel IOM task), an error handling task (called the pageable-channel IOM task), and an OU task. The queues are the resident-channel IOM queue, the OU queue, and the channel communications queue (for communications among the resident channel IOM task, the pageable channel IOM task and the machine services control point).

The channel event processing function performs the following:

- Receives messages at vary-on time from machine services control point (MSCP). These messages contain the QCT offset, OU number, channel priority, and IOM queue address for the device being varied on. This information is used to update internal tables that are used during error event processing, FOB timing, and OU task failure processing to identify active OU numbers and locate the associated QCTs and IOM queue addresses.
- Receives messages at vary-off time from the MSCP. These messages are returned and any information about the device is removed from the channel error function internal tables.
- Sends read-event and start-channel commands to the OU queue for processing by the OU task. These commands cause the OU task to return a channel hardware error event, a post event, or an event handler error event.

- Responds to events as follows:
 - A start channel command is issued if an event handler error with channel secondary error event is received.
 - All events except channel secondary error events are logged.
 - For errors on devices identified in the internal tables, the appropriate IOM(s) is notified using a channel error message.

- Provides an FOB time-out function to notify the device IOM when a certain number of time intervals have passed without the FOB being completed. The device IOM indicates how many time intervals are allowed by setting the FOB time limit field in the QCT. The time interval is 30 seconds. The FOB time-out function does not apply to storage management devices. The device IOM is notified through a channel-error message of any FOB that has suspended activity.

- Checks a counter in the source/sink active device list for OU tasks that have encountered a hardware exception. In this case, the channel event processing function sends a channel error message to the associated device IOM queue. This message notifies the device IOM of the OU task failure.

- Provides support for the Power Warning feature. A power warning event is signaled and the resident channel IOM sends a message to the SCA IOM for a full uninterruptible power supply or to the mini-uninterruptible power supply task for a mini-uninterruptible power supply.

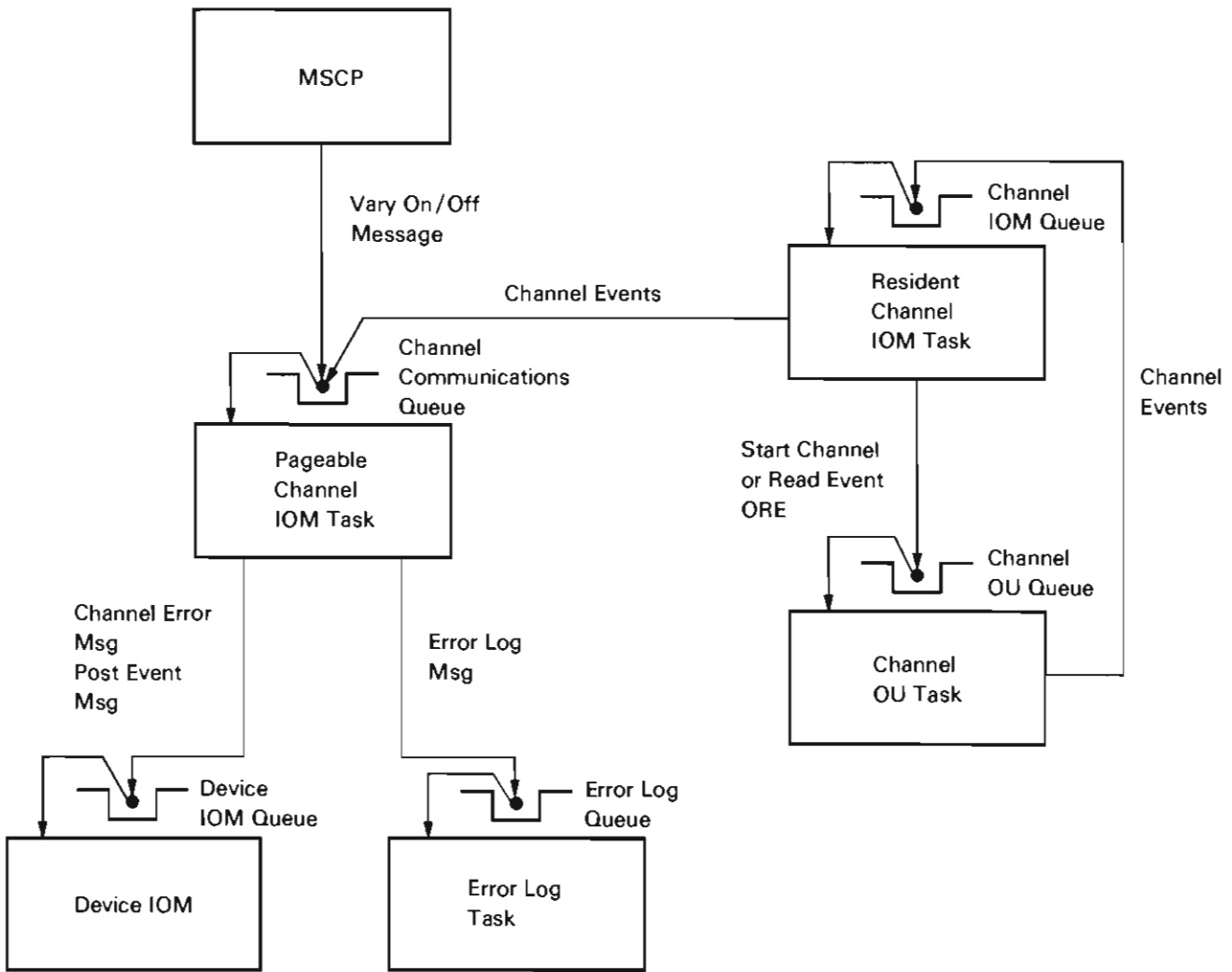


Figure 21-2. Channel Event Processing Function

Resident Channel IOM

#LOCHRCI performs channel event processing for functions that cannot wait for paging and to correct errors that prevent paging. #LOCHRCI is resident in the nucleus and does not use other functions that require paging. #LOCHRCI performs the following functions:

- Removes events (except for event handler errors with channel secondary error and power warning events) from the channel event processing QCT event list and passes the events using channel communication messages to the pageable channel IOM.
- Processes event handler errors with channel secondary errors. When these errors are received, #LOCHRCI issues a start channel command to restart the channel and enable paging to occur.
- Processes power warning post events by sending a message to the SCA IOM for full UPS or the basic-UPS task for basic-UPS.

Pageable Channel IOM

#LOCHPCI performs channel event processing for functions that can or must wait for paging. #LOCHPCI performs the following functions:

- Interfaces with synchronous MSCP routines at vary-on and vary-off times.
- Responds to channel communications messages from the resident channel IOM.
- Provides FOB timing.
- Checks for OU tasks with a microprogramming exception.

At vary-on time, a message is received from the MSCP. This message contains the QCT offset, OU number, channel priority, and IOM queue address for the device being varied on. #LOCHPCI stores this information in an internal table and returns the message to the MSCP. When a vary-off message is received from the MSCP, #LOCHPCI removes the information corresponding to the device being varied off from the internal table and returns the message to the MSCP.

#LOCHPCI responds to channel communications messages from the resident channel IOM as follows:

- Determines the event type (post event attention, post event error, channel hardware error, event handler error, and unlogged event counts).
- Determines if the OU number is known and if an entry for the OU number exists in the internal table. If an entry exists, #LOCHPCI sends a channel error message to the device IOM queue.
- For adapter-wide post event errors, #LOCHPCI sends a channel error message to each device IOM queue for that channel priority.
- Logs the event and the time the event was received.

The device IOM requests FOB timing by setting the FOB limit field in the QCT. At fixed time intervals (every 30 seconds), #LOCHPCI scans the FOBs being timed. If the FOB has not completed, #LOCHPCI updates a counter in the QCT. If the value in the counter exceeds the limit set by the device IOM, #LOCHPCI sends a channel error message to the device IOM.

If an OU task causes a microprogramming exception, the exception handler executes a Receive Count instruction to a counter in the source/sink active device list. #LOCHPCI checks this counter every 30 seconds. If a task is found waiting on the counter, the internal device tables are checked for an IOM queue address that matches the address of the task waiting on the counter. If a match is found, the OU task and the machine communication area are dumped to VLOG through the channel event queue. Then a channel error message is sent to the device IOM queue to inform the device IOM of the OU task failure.

DATA AREAS

Channel Error Message

The channel IOM notifies the device IOMs of channel errors and FOB time-outs by sending a 40-byte message to the device IOM queue. The message consists of a 24-byte header (ZZSSVHDR), an 8-byte time stamp, and a 4-byte channel event or an 8-byte VLOG identification (ZZLOCHE1).

Channel Vary On/Off Message

The channel vary on/off message consists of a 24-byte header (ZZSSVHDR), a 6-byte IOM queue address, a 6-byte QCT address, a 1-byte OU number, and a 1-byte channel priority (ZZLOCHVO).

STRUCTURE

The following is a list of the modules in the channel IOM and the function that each module performs. The list also shows how the module is invoked.

#LOCHPCI Pageable Channel IOM

Function: Performs channel event processing for functions that can or must wait for paging.

How Invoked: Within this component.

#LOCHRCI Resident Channel IOM

Function: Performs channel event processing for functions that cannot wait for paging and for errors that prevent paging. This module is part of the resident nucleus.

How Invoked: Other VMC components.

#LOSHIOM Start/Halt Device

Function: Validates input, and then builds and sends a start or halt device ORE/FOB to the OU queue for processing by the OU task.

How Invoked: Other VMC components.



Error Log

INTRODUCTION

The error log provides a method for other VMC routines to log errors and retrieve the logged entries. The interface to the error log is a send/receive interface through the error log input queue.

To log an error, the requesting VMC routines send a logging request message to the error log input queue. The logging request contains identification information, a time stamp, and a pointer to the error data. The error log component receives the message and files it in the time sequence with all other entries that have the same identification.

To retrieve logged entries, the requesting VMC module sends an error log retrieval request message to the error log input queue. The retrieval request message identifies the entry to be retrieved and the area to receive the entry, and the attributes of the search to be performed by the error log component. The error log component receives the request, locates the entry, and returns the entry to the area specified by the requesting VMC routine.

After the requester has received the first message in the desired time frame, the requester requests the next entry for this identifier (using flags in the message) and continues until all messages have been returned or the time stamp returned is out of the specified range.

The requester can request data from the error log using ignore identifier mode. This allows the requester to build a list of identifiers or detect any unexpected entries.

Note: Time sequencing is lost in ignore identifier mode.

The error logging function (`#MSERRLG`) is an asynchronous task used to log error messages. A request to log a message is made by sending a special message to a queue (pointed to by `MCA4VELQ` in the machine communications area). The message points to a standard log request, that in turn, points to requester specified data.

DATA AREAS

Error Log Request

An overview of the error log request is shown in Figure 22-1. This is a message used to request that an entry be inserted into or retrieved from the error log.

Header (ZZSSVHDR) <ul style="list-style-type: none">• Function• Pointer to Reply Queue
Error Log (ZZMSELOG) <ul style="list-style-type: none">• Pointer to Data Area• Time Stamp• Lengths and Attributes• Error and Device• Identification

Figure 22-1. Error Log Request

Error Log

Figure 22-2 shows the layout of an error log entry and the identification system. The figure shows the logical form of the log; the format shown in the figure is not necessarily the format in which the entries are passed to or from the error log routine.

Identifier Type	OU Number	Exchange Identification	Time Stamp	Sequence Number	Record Type	Error Class	Error Code	Data
-----------------	-----------	-------------------------	------------	-----------------	-------------	-------------	------------	------

Figure 22-2. Error Log Entry

STRUCTURE

The error log functions are performed by a single module (#MSERRLG). The following shows the function of this module, and how this module is invoked.

#MSERRLG Error Log

Function: Receives a request from the error log queue, processes the request, and returns the request to the appropriate response queue.

How Invoked: Other VMC components (primarily I/O managers) to log errors. The error recording edit program retrieves and formats the entries for display or printing.

Instruction Processors

INTRODUCTION

The instruction processors directly support the System/38 source/sink instructions. There is an instruction process for each of the 13 source/sink instructions. These processors operate against the following primary object types:

- Logical Unit Descriptions (LUDs)
- Controller Descriptions (CDs)
- Network Descriptions (NDs)
- Request I/O Response Queue

The instruction processors are used to create, materialize, modify, and destroy the preceding objects, and to perform I/O operations. The instruction processors are invoked through the supervisor link-supervisor exit (SVL-SVX) linkages whenever the corresponding System/38 instruction is executed. The instruction processors execute synchronously with the user procedure; however, the instruction processors can cause asynchronous processing by invoking other VMC tasks. Task switching from the instruction processors to other VMC tasks (for example, the I/O managers) is always initiated through a send-message operation.

Create Instruction Processors

The create instruction processors perform the create ND, CD, and LUD functions. There is a separate processor for each create instruction. These processors create the specified object according to the input template operand in the instruction.

Materialize Instruction Processors

The materialize instruction processors perform the materialize ND, CD, and LUD functions. There is a separate processor for each materialize instruction. These processors materialize the specified object according to the materialization option specified in the option operand in the instruction. The materialization option allows the entire object, individual elements within the object, or groups of elements to be selected for materialization.

Modify Instruction Processors

The modify instruction processors perform the modify ND, CD, and LUD functions. There is a separate processor for each modify instruction. These processors modify the specified object according to the specification in an input template and modification operands in the instruction. The modification option allows the entire object, individual elements within the object, or groups of elements to be modified.

Destroy Instruction Processors

The destroy instruction processors perform the destroy ND, CD, and LUD functions. There is a separate processor for each destroy instruction. These processors destroy the specified object and free the storage space used by the object.

Request I/O Instruction Processor

The Request I/O instruction processor schedules the requested work to the appropriate device IOM, to the load/dump processor, or to machine service control point (MSCP) by building a message to be sent to the input queue of that component. The Request I/O instruction is the only path from the machine interface to the source/sink devices. The operation performed by the IOM and the return of the feedback record are not functions of the Request I/O instruction processor; the function of the Request I/O instruction processor is complete when the message is sent to the IOM component. Figure 23-1 shows the relationships among the Request I/O instruction processor and the IOM components for a normal Request I/O instruction. For a synchronous Request I/O instruction, the I/O is completed before control is returned to the user program.

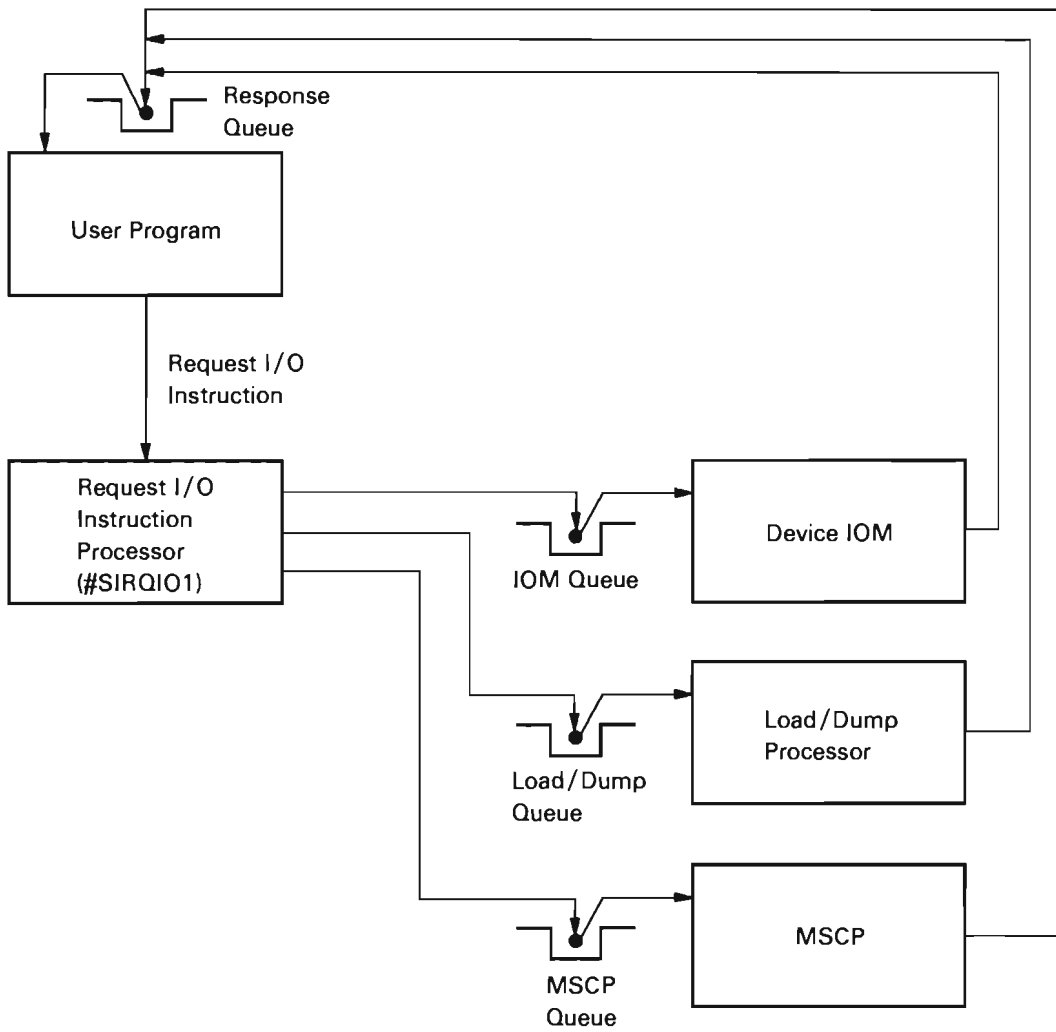


Figure 23-1. Request I/O Overview

DATA AREAS

Also refer to the *Vertical Microcode Overview* section of this manual for additional descriptions of source/sink data areas.

Logical Unit Description (ZZSILUOB)

The LUD describes a physical I/O device attached to the system. The LUD contains:

- Segment group header
- Encapsulated program architecture (EPA) header
- Type definition data
- Pointers and internal data
- Physical definition data
- Session information
- Load/dump information
- Status indicators
- Internal VMC work area
- Specific characteristics
- Retry value sets
- Error threshold sets
- Device-specific contents

Controller Description (ZZSICDOB)

The CD represents a physical controller. The CD also links the ND and the LUD. The CD contains the following:

- Segment group header
- EPA header
- Type definition data
- Pointers and internal data
- Physical definition data
- Dial digits
- Station control information
- Activate physical unit information
- Selected mode data
- Status indicators
- ND candidate list
- Specific characteristics
- Transmit identification data area
- Unit-specific contents

Network Description (ZZSINDOB)

The ND describes an I/O port and teleprocessing line for remotely attached I/O devices. The ND contains the following:

- Segment group header
- EPA header
- Type definition data
- Pointers and internal data
- Communications initialization information
- Exchange identification
- Physical definition data
- Communications subsystem parameter data
- Line definition data
- Selectable mode data
- Status indicators
- CD eligibility list
- Specific characteristics
- Retry value sets
- Line-specific contents

OU/ND Table (ZZSSOUND)

The OU/ND table is used to keep track of how many NDs exist and which NDs are varied on or in diagnostic state. The OU/ND table contains the following:

- Send/receive count for serialization
- Number of OU/ND entries (12)

For each OU number, the OU/ND table contains the following:

- OU number
- Flag bits indicating if first or last OU number within a group
- Count of NDs in existence (maximum of 10)
- Number of NDs varied on (maximum of 1)
- Number of NDs in diagnostic state (maximum of 1)
- Pointer to varied-on ND
- Pointer to ND in diagnostic state

STRUCTURE

The following is a list of the source/sink instruction processor modules and the function that each module performs. The list also shows how the module is invoked.

#SICDCR1 Create CD

Function: Validates the input template and creates a controller description according to the template.

How Invoked: Create Controller Description instruction.

#SICDDY1 Destroy CD

Function: Destroys the specified controller description.

How Invoked: Destroy Controller Description instruction.

#SICDMD1 Modify CD

Function: Modifies the specified controller description element(s) according to the modification options and input template operands.

How Invoked: Modify Controller Description instruction.

#SICDMT1 Materialize CD

Function: Materializes the specified controller description element(s) according to the materialization options operands.

How Invoked: Materialize Controller Description instruction.

#SICDRT1 Range Table for CD

Function: Performs range table checking on various elements of the creation or modification input template operands.

How Invoked: Within this component.

#SILUCR1 Create LUD

Function: Validates the input template and creates a logical unit description according to the template.

How Invoked: Create Logical Unit Description instruction.

#SILUDY1 Destroy LUD

Function: Destroys the specified logical unit description.

How Invoked: Destroy Logical Unit Description instruction.

#SILUMD1 Modify LUD

Function: Modifies the specified logical unit description according to the modification options and input template operands.

How Invoked: Modify Logical Unit Description instruction.

#SILUMT1 Materialize LUD

Function: Materializes the specified logical unit description element(s) according to the materialization options operands.

How Invoked: Materialize Logical Unit Description instruction.

#SILURT1 Range Table for LUD

Function: Performs range table checking on various elements of the creation or modification input template operands.

How Invoked: Within this component.

#SINDCR1 Create ND

Function: Validates the input template and creates a network description according to the template.

How Invoked: Modify Network Description instruction.

#SINDDY1 Destroy ND

Function: Destroys the specified network description.

How Invoked: Destroy Network Description instruction.

#SINDMD1 Modify ND

Function: Modifies the specified network description element(s) according to the modification options and input template operands.

How Invoked: Modify Network Description instruction.

#SINDMT1 Materialize ND

Function: Materializes the specified network description element(s) according to the materialization options operands.

How Invoked: Materialize Network Description instruction.

#SINDRT1 Range Table for ND

Function: Performs range table checking on various elements of the creation or modification input template operands.

How Invoked: Within this component.

#SIRQI01 Request I/O Processor

Function: Processes the Request I/O instruction and builds a message that is placed in the queue of the appropriate VMC component.

How Invoked: Request I/O instruction.

#SIRQSYN Synchronous Request I/O Processor

Function: Processes a Synchronous Request I/O instruction. When I/O is complete, control is returned to the user program. No feedback record remains to be dequeued.

How Invoked: Within this component.

#SSINOUT Initialize OU/ND Table

Function: Creates and initializes the OU/ND table.

How Invoked: Other VMC component.

Synchronous Data Link Control Primary and Secondary I/O Managers

INTRODUCTION

An SDLC I/O manager (IOM) activates, manages, and de-activates the telecommunications channel and enforces the synchronous data link control (SDLC) protocol. A SDLC primary IOM task is created by the machine services control point (MSCP) as a result of a Modify Network Description (vary on) instruction. System/38 provides separate tasks for primary and secondary SDLC roles.

An SDLC IOM is used for communications to devices on a switched point-to-point line, a nonswitched multi-point line, or a nonswitched point-to-point line and interfaces with the following:

- Machine services control point (MSCP)
- Channel IOM
- Modify Network Description instruction processor
- Modify Controller Description instruction processor
- Transmit and receive operational unit (OU) tasks

The user of the SDLC IOM can execute Modify Controller Description and Modify Network Description instructions to establish the SDLC data link with the remote end. The SDLC IOM ensures that the SNA frames reach their destination on the SDLC data link. One SDLC secondary IOM is required for each telecommunications line to which System/38 is connected.

- Station IOMs
- The error log
- An I/O controller (IOC)

A SDLC IOM task is associated with one IOC line position as shown in Figure 24-1. The IOC simultaneously supports four teleprocessing lines and interfaces with the four corresponding line IOM tasks.

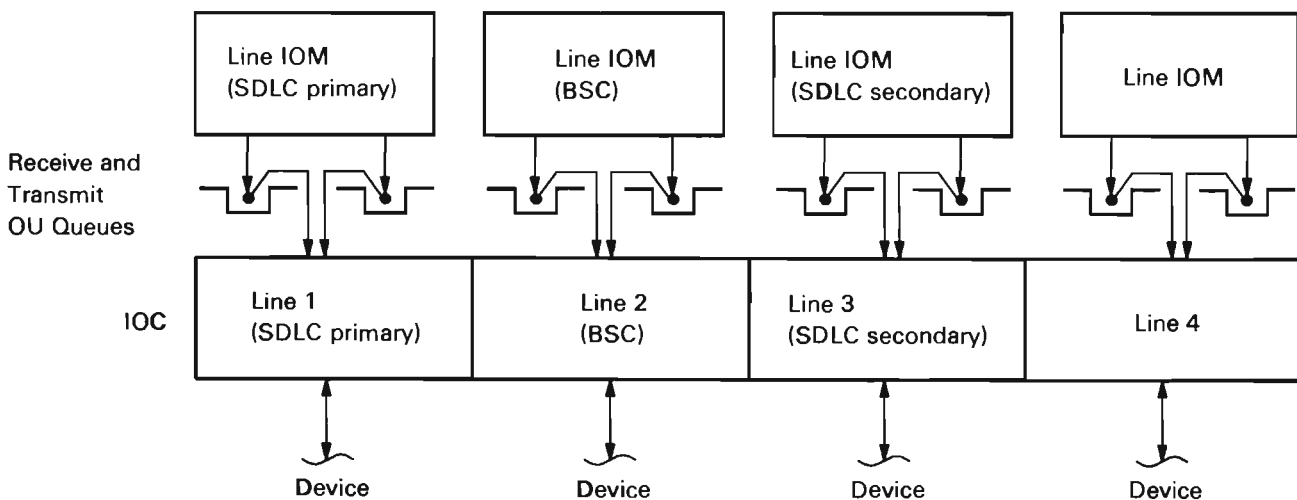


Figure 24-1. Line IOM/IOC Line Position Relationships

The SDLC IOM is a VMC task created by MSCP when a Modify Network Description (vary on) instruction is issued against an SDLC network description (ND) object. The task is created with one input queue upon which the send/receive messages and the operational response elements (ORE) are placed. The MSCP provides the SDLC IOM with transmit and receive operational unit tasks to communicate with the communications input/output controller (IOC). The MSCP also invokes the communications IOC wake up module (if necessary) to initialize the IOC.

System/38 Instruction Support

All source/sink modify instructions passed to the SDLC IOMs are in the form of messages (SRMs). These messages are built by mapping the instructions to the VMC format. This step is performed by the source/sink instruction processors.

All messages are routed to the appropriate modules by the SDLC IOM routers. The instructions supported are described in the following paragraphs.

Modify Network Description (Vary-On/Off): These instructions establish or break the interface between the secondary SDLC IOM and the communications I/O controller (IOC), and initialize or reset the IOC for the telecommunications line.

Modify Network Description (Manual Answer/Abandon Call): This instruction synchronizes the hardware adapter signals with the operator actions for completing or discontinuing a switched manual answer connection.

Modify Network Description (Continue/Cancel): This instruction is used to allow the reuse or to suspend the reuse of the network description after a nonrecoverable error.

Modify Network Description (Start Data): This instruction indicates that the operator has manually placed the coupler in data mode, and the line is now ready for data communications.

Modify Controller Description (Vary-On/Off): These instructions are used by device management to establish or to break the communications path to the system services control point (SSCP) in the primary unit or station represented by the controller description from the physical unit (PU) in the secondary System/38.

Modify Controller Description (Dial): This instruction allows a station to be dialed manually or automatically.

Modify Controller Description (Abandon Connection): This instruction disconnects the secondary SDLC IOM from a remote station. The secondary SDLC IOM stays active; but the switched line is disconnected.

Modify Controller Description (Continue/Cancel): These instructions allow the reuse or suspend the reuse of the device after an unrecoverable error.

Communication to a SDLC IOM is through a send/receive message, which the SDLC IOM receives through a single send/receive queue as shown in Figure 24-2. The message can be generated by either another VMC function or an operation request element (ORE).

The queue message router receives the message and uses the key field in the message to determine if the message is a VMC message or an ORE. Then, based on the function field, the queue message router invokes the appropriate message or ORE handler as shown in Figure 24-3 for a message, and Figure 24-4 for an ORE.

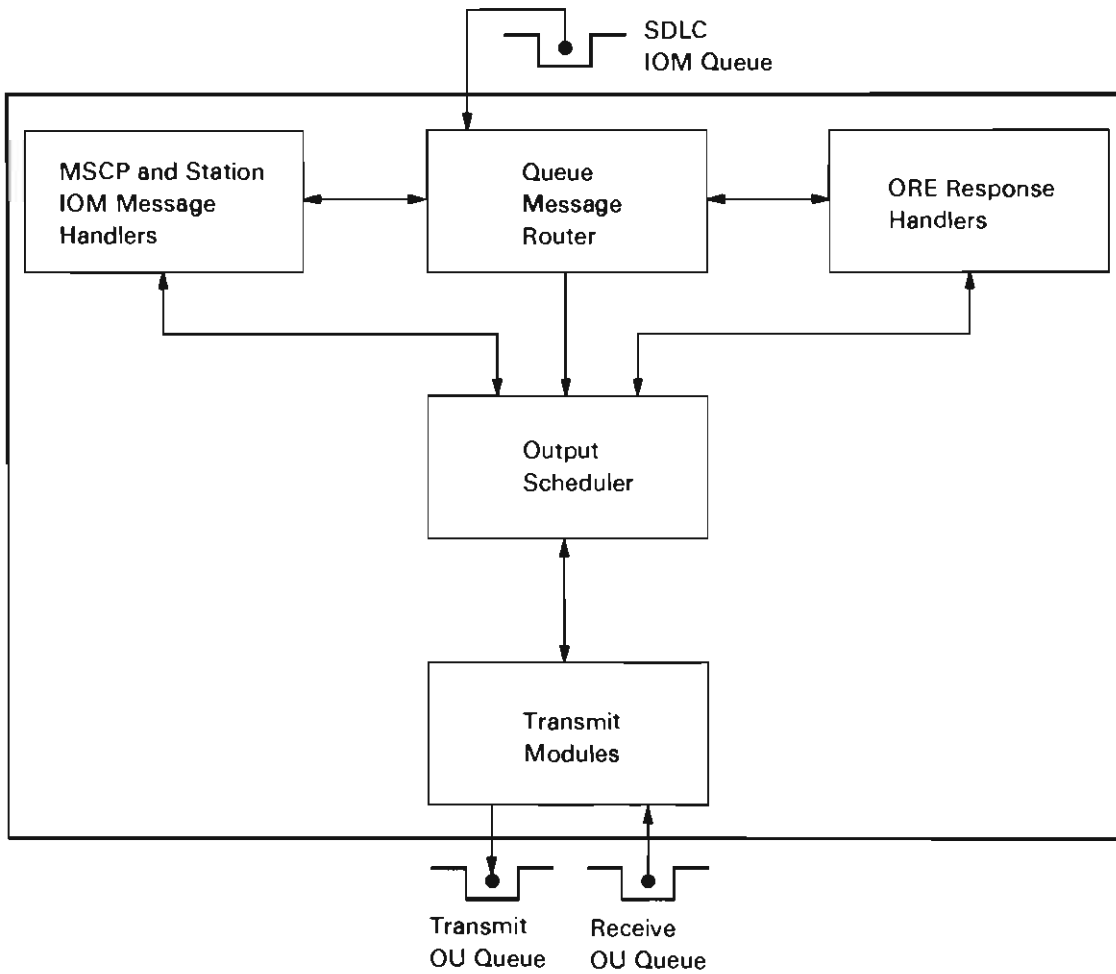


Figure 24-2. SDLC IOM Processing

Function	Requester	SDLC Primary IOM Module	SDLC Secondary IOM Modules
Activate link	MSCP	#TPLALK0	#TP2ALK0
De-activate link	MSCP	#TPLDAK0	#TPLDAKO
Request exchange identification	MSCP	#TPLXID0	#TP2RSID
Contact	MSCP	#TPLCNT0	#TP2CNT
Discontact	MSCP	#TPLDCT0	#TP2DCT
Initialize line	MSCP	#TPLINLN	#TPLINLN
Activate connect-in	MSCP	#TPLACIN	#TPLACIN
De-activate connect-in	MSCP	#TPLDCIN	#TPLDCIN
Connect-out	MSCP	#TPLCOUT	#TPLCOUT
Abandon connect-out	MSCP	#TPLACOT	#TPLACOT
Abandon connection	MSCP	#TPLABCN	#TPLABCN
Output request	Station IOM	#TPLQRT0	#TP2SDLC
Execute test	Diagnostic	#TPLEXT0	N/A
Read data store	Diagnostic	#TPLRDS0	#TP2RDSO
Internal trap	Diagnostic	#TPLTRP0	#TP2TRPO

Figure 24-3. Message Function Handlers

Function Key Field	Type	Command	SDLC Primary IOM Module	SDLC Secondary IOM Modules
	POB	Write data	#TPLQRT0	#TP2SDLC
	FOB	Set line priorities	#TPLSLPR	#TPLSLPR
	FOB	Initialize	#TPLALK1	#TP2ALK1
	FOB	Line reset	#TPLRSTR	#TPLRSTR
	FOB	Enable switched connection	#TPLESCR	#TPLESCR
	FOB	Auto dial	#TPLAUDL	#TPLAUDL
	FOB	Read sense	#TPLIOL1	#TPLIOL1
	FOB	Write data	#TPLWDNS	#TP2SDLC
XID	FOB	Read data	#TPLXID2	#TP2XIDR
SNRM	FOB	Read data	#TPLCNT2	#TP2SNRM
DISC	FOB	Read data	#TPLDCT2	#TP2DM
Information/ Supervisory frame	FOB	Read data	#TPLISIN	#TP2ISIN
Test	FOB	Read data	#TPLTST2	#TP2TSTR
	FOB	Read data store	#TPLRDS2	#TP2RDS2
	FOB	Set monitor mode	#TPLTRP2	#TP2TRP2
¹ These modules are invoked from #TPRRDDA.				

Figure 24-4. ORE Function Handlers

The output scheduler shown in Figure 24-2 is invoked by the message handlers. The output scheduler scans the SOT until all entries have been checked. When an entry with a pending status is found, the output scheduler invokes the appropriate transmit module for the specified status as shown in Figure 24-5.

The error recovery procedure modules in the SDLC IOM centralize recovery procedures for line IOM- and IOC-detected errors. The error recovery procedure modules are invoked by the ORE response handler when an error is detected. The queue message router also detects unrecoverable errors and invokes the appropriate error recovery procedure module.

Status/Condition	SDLC Primary IOM Module
XID Pending	#TPLXID1
SNRM Pending	#TPLCNT1
DISC Pending	#TPLDCT1
Output Pending	#TPLSCED
Test Pending	#TPLTST1
Confirmation due or contacted without output pending	#TPLSCED
Skip	---
Response Pending	---
Read Data Store Pending	#TPLRDS1
Internal Trap and Set Monitor Mode Pending	#TPLTRP1

Figure 24-5. Transmit Modules

Connect

Before any line on a communications IOC is activated, the IOC control storage must be initialized by an MSCP function of VMC. Following this function, the MSCP creates a SDLC IOM task and a queue. The SDLC IOM task is entered at the queue message router and executes a receive message instruction to the queue. The SDLC IOM then awaits an activate link message to be sent by the MSCP.

When the activate link message is received, the queue message router invokes the activate link handler to handle the message. For SDLC primary the backward object chain of the network description (ND) is used to determine the number of stations and the largest required input buffer size. The activate link handler then acquires machine-wide storage, initializes the service order tables, input buffers, and FOBs. Next, an ORE is built to send the set line priorities command. The line priority field is passed by the MSCP as a parameter in the message. Control is then returned to the queue message router, which again executes a receive message instruction. When the set line priorities ORE is returned, it is routed to #TPLSLPR, which either builds and sends a line reset ORE for a switched line or invokes #TPLINLN (initialize line) for a nonswitched line.

Initialize line builds an initialize ORE based on parameters in the network description. After the initialize ORE has been returned, it is routed to #TPLALK1 which sets the status field of the activate link message and executes a send message instruction to return the message to the response queue.

For nonswitched lines, the line activation operation is completed. For switched lines, the MSCP sends either an activate connect-in message for autoanswer, manual answer, and manual dial, or an activate connect-out message for autodial.

When the activate connect-in message is received, the queue message router invokes #TPLACIN, which builds and sends an enable switch connection ORE. The response to the enable switched connection ORE is processed by module #TPLESCR, which sets the status in the activate connect-in message and returns it to the MSCP.

When a connect-out message is received, #TPLCOUT is invoked to build and send an autodial ORE. The response to the autodial ORE is processed by #TPLAUDL, which returns the connect-out message.

Because the line IOM always returns to the receive message state of the queue message router, the MSCP can interrupt previous switched-connection operations. To halt a connect-out (autodial) operation, the MSCP sends an abandon connect-out message. This message is processed by #TPLACOT, which builds and sends a line reset ORE to the receive OU queue. This command resets the IOC and associated hardware and causes the autodial ORE to be returned. An enable switched connection (activate connect-in) is interrupted by a de-activate connect-in from the MSCP. This message is processed by #TPLDCIN. In this case the line reset ORE is sent to the transmit OU queue. When the enable switched connection or autodial ORE is returned, the queue message router determines that a line reset was issued and does not invoke #TPLESCR or #TPLAUDL.

XID for SDLC primary

Once the connection is established, whether the line is switched or nonswitched, the MSCP directs the initial communication with the station using the request exchange station identification message. This message is routed to #TPLXID0 which, for nonswitched lines, locates the appropriate service order table entry for the text in the message. For a switched connection, the first SOT entry is used because the identity of the station has not been established. An XID control area is established and the XID frame is transmitted within it.

The request exchange identification message is returned to the MSCP, the SOT entry status is set to exchange identification pending, and the output scheduler is invoked. The output scheduler searches the SOT entries for pending output. Because exchange identification pending is set for an entry, #TPLXID1 is invoked. This module builds the Read Data command in the appropriate ORE in the link control block (LKB). It also sets exchange identification sent and the station address in the ORE key. An input buffer is located by searching the buffer control list for an entry with free status. Finally, the read data ORE is sent to the receive OU queue (the IOC microcode maintains the read pending until a poll bit is sent by a write data ORE).

Next, a write data ORE is built using the same ORE key function as the read data. The FOB data address is that of the transmit XID in the XID control area of the SOT entry. The first byte is set to the appropriate SDLC address (specific address for nonswitched, all stations for switched), the second byte is set to the SDLC nonsequenced command exchange identification with the poll bit set.

The write data ORE is then sent to the transmit OU queue. The ORE key is set with a function and station address so that error recovery can properly associate the station and the error when the ORE is returned.

Finally, the transmit busy and receive busy LKB status bits are set. Control is returned to the output scheduler, which finds transmit and receive busy set, discontinues the search of the SOT, and returns control to the queue message router. When the write data ORE is returned, the queue message router invokes #TPLWDNS, which handles write data ORE responses for nonsequenced or supervisory type SDLC commands.

After the basic/functional status (BSTAT/FSTAT) indicators are checked for proper completion, transmit busy is reset and control is returned to the queue message router. When the read data ORE is received, the queue message router uses the FOB command and the ORE key function to invoke #TPLXID2, which handles the read data ORE and exchange identification response.

If a disconnect mode response is received or if the BSTAT/FSTAT indicates an idle state detect time-out, the SOT entry is checked to see if delayed contact is allowed for this station. (Delayed contact allows the controller description to be varied on before the station is physically powered on.) If delayed contact is allowed, a wait time-out is issued for approximately 60 seconds, and the skip bit is turned on.

When the wait time-out message (issued when the time-out expires) is received from the response queue, it is directed to #TPLDELY. This module searches the entire SOT for entries that are in delayed contact status and, upon finding such an entry, resets the skip bit so the exchange identification will be attempted again when the scheduler next services that station. Inactive controllers in delayed contact status are inserted into the polling list one at a time so it requires a complete cycle through the polling list for each ('N') controller in delayed contact status when the timer expires. This ensures that no active controllers will have their response time impacted by more than one idle state time value in any one polling cycle through the SOT table and that no inactive controller has to wait longer than one delayed contact time interval plus ('N') polling cycles to be allowed back on the line. The timer is started when all inactive controllers specified as delayed contact have been retried.

If the BSTAT/FSTAT indicates error-free completion, a request contact message is sent to the MSCP. Error-free completion consists of validation of the ORE key station address, validation of the SDLC address in the buffer (nonswitched), and assurance that the station response was a valid exchange. The text of this message contains the exchange identification information received from the station.

The SOT entry status is set to skip, the LKB receive busy status is reset, and control is returned to the queue message router. When the request contact message is returned by the MSCP, the response bit is set in the VMC header function field. This bit indicates to the queue message router that the module #TPLSRMR is to be invoked. #TPLSRMR handles the returned message and, in the case of request contact, inoperative, error log, and input request, frees the machine-wide storage containing the message.

XID3 Format Checking

When an XID3 is received, a request contact VMC SRM with error status is sent to the MSCP for any of the following:

- The received XID is too short for a valid XID.
- An error control vector is found in the received XID.
- The received XID does not specify a negotiable SDLC role and contains values unacceptable to the SDLC IOM. An XID3 with error appended is transmitted to notify the other end of the unacceptable value.

A result field is updated in the XCA to inform the calling program of the format check results.

The receipt of any control vector in the XID other than a network name control vector or an error control vector causes the search for an error control vector to be discontinued.

Contact for SDLC Primary

The operation for contact is the same as that described under *Connect* with the following exceptions:

- The following modules are substituted:

Normal Flow	Contact
#TPLXID0	#TPLCNT0
#TPLXID1	#TPLCNT1
#TPLXID2	#TPLCNT2

- #TPLCNT2 does not set the skip bit in the SOT entry status.
- A request contact is not sent to the MSCP.
- An SDLC nonsequence command Set Normal Response Mode command is sent and a nonsequenced acknowledgment is received.

Contact for SDLC Secondary

When a read ORE returns a SNRM command, #TP2SNRM is called to place the station (SOT entry) in normal response mode. If the station is already in normal response mode and an intervening S- or I-frame is received, a second SNRM VMC SRM is sent to the SIOM. If a station is contacted, it is reset and 'contact pending' is set.

In the SOT entry, Ns and Nr counts are zeroed out and a frame reject state indicator is cleared. The response transmitted is determined by the following:

- If disconnect is pending, an SDLC RD is sent.
- If contact is pending, an SDLC DM is sent.

- If the XCA does not identify the other end to be a Type 2.1 node, an SDLC UA response is sent. In this case, a request contact VMC SRM is not sent to the MSCP. When the host sends an activate physical unit SNA request, common SNA path control is invoked to send a request activate physical unit SRM to the MSCP. If the request activate physical unit response is positive, it is sent to the SIOM. If the response is negative, common SNA support is invoked to transmit a negative activate physical unit response.
- If the XCA identifies a Type 2.1 node and the SIOM is contacted, an SDLC UA is sent.
- If the XCA identifies a Type 2.1 node and the SIOM is not contacted, an SDLC DM is sent and a request contact SRM is sent to the MSCP. The text of the message contains the last XID received and 'contact pending' is set.

Discontact

The operation for discontact is the same as that described under *Connect* with the following exceptions:

- The following modules are substituted:

Normal Flow	Discontact
#TPLXID0	#TPLDCT0
#TPLXID1	#TPLDCT1
#TPLXID2	#TPLDCT2

- A request contact is not sent to the MSCP.
- The SDLC nonsequenced Disconnect command is sent.
- The SOT entry is cleared and the entry status is set to skip.

Test

The operation for test is the same as that described under *Connect* with the following exceptions:

- The following modules are substituted:

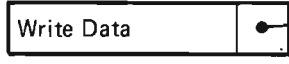
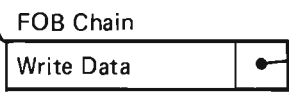
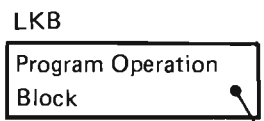
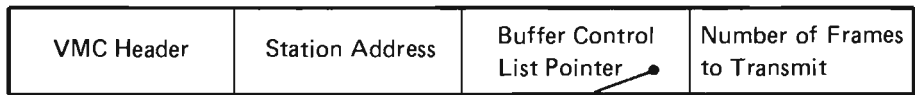
Normal Flow	Test
#TPLXID0	#TPLEXT0
#TPLXID1	#TPLTST1
#TPLXID2	#TPLTST2

The input and output buffers are supplied by the requester, and pointers to these buffers are contained in the message text.

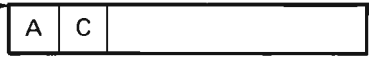
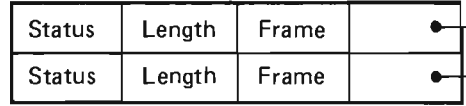
Normal Flow

Once contact processing is complete, the SOT entry status is set to contacted. This indicates that when the output scheduler encounters this entry, it transmits a supervisory frame to poll for input.

However, as the station IOM task is activated, the SNA sessions are established and the request/response units are exchanged between the primary and the secondary station. In order to cause the SDLC IOM to transmit I- (information) frames, an output request message is sent to the SDLC IOM queue. The queue message router invokes the output request handler. This module sets output pending status in the SOT entry and invokes the output scheduler. The output scheduler encounters the SOT entry, finds the output pending status, and invokes the I-frame transmission handler. This module retrieves the output request message, locates the buffer control list of the station IOM, and, using the number-of-frames value in the message, begins to build the program operation block/function operation block operational program. Each frame submitted by the station IOM is addressed by a separate FOB that contains a Write Data command as shown in Figure 24-6. The SDLC control field of each frame is set to indicate the SDLC number sent and number received. An input buffer is then obtained by using the SDLC IOM buffer control list, and a read data ORE is built and sent to the receive OU queue. The operation request element/program operation block is then sent to the transmit OU queue. Finally, transmit and receive busy status indicators are set in the LKB, and control is returned to the output scheduler.



Station IOM Buffer Control List



¹ Poll bit is set.

Figure 24-6. Output Request-Operational Program Relationship

The queue message router invokes the I-frame write response handler when the operation request element/program operation block is received. This module checks both the BSTAT and the FSTAT of each FOB for completion. For a complete transmission, transmit busy in the LKB is reset.

When the read data ORE is received, the queue message router or the read data analysis module invokes the I-frame write response handler. (The function field in the ORE key had been set to information/supervisory frame sent.) The I-frame write response handler determines which frames have been received (provided BSTAT/FSTAT or ISTAT indicates no errors). For a supervisory response, the number received by the station is compared with the number sent by the system. If in agreement, the output request message is retrieved, the frame positions in the buffer control list of the station IOM are set to completion status, the VMC header status is set, and the message is returned to the station IOM. Output pending status in the SOT entry is reset.

For I-frame(s) received, the number sent/received comparison is made as in the preceding description. Each ISTAT is checked, the number sent is incremented, and an input request message is built and sent to the station IOM.

Finally, receive-busy in the LKB is reset and control is returned to the output scheduler. The next time this station is selected by the output scheduler, the I-frames received are confirmed to the station. This occurs through either a supervisory frame or additional I-frames (if another output request message is received). The line IOM then continues to service the station(s) until all SOT entry status indicators are set to skip.

SDLC Autopoll Flow

SDLC autopoll cannot be performed if there are more than eight stations varied on the SDLC primary line.

#TPLAUTO directs the autopoll function. This module is invoked by the scheduler when the following conditions are met:

- There has been no activity except polling on the line and no station on the line has responded with I-frames.
- There are not more than eight stations varied on.

#TPLAUTO updates the control byte for each entry in the poll list according to the send and receive counts in the SOT entry, builds the autopoll transmit ORE and a receive ORE, and sends the OREs to the appropriate OU task queues. The transmit ORE is posted after one pass through the poll list; the receive ORE is returned when the autopoll operation is terminated by one of the following conditions:

- A polled station changes state from receive ready to receive not ready.
- A polled station changes state from receive not ready to receive ready.
- A polled station responds with other than receive ready or receive not ready.
- A polled station responds with an unexpected sequence count.
- A Stop Autopoll command is issued that returns the receive ORE associated with the Autopoll command.
- A polled station responds with one or more bytes after the control byte.
- A polled station does not respond.
- Any error condition is detected.

Vary Off

When the MSCP is directed to vary off a line, it sends a de-activate link message to the SDLC IOM queue. The queue message router invokes #TPLDAKO, which issues a halt to both OU tasks, frees the machine-wide storage obtained by the activate link handler, and remains in a receive message loop until all messages sent by the SDLC IOM are returned. The last instruction is a destroy task function that destroys the primary SDLC IOM task.

Error Flow

A substantial portion of the SDLC IOM is involved in the recovery of the many errors that are possible in the management of the telecommunications channel. These errors are categorized as follows:

Level	Recovery	Example
LINK	Permanent Error, No Retry	BSTAT0 = I/O Error BSTAT1 = Line Not Initialized
LINK	Retryable	BSTAT0 = I/O Exception BSTAT1 = Clear to send Inactive
STATION	Permanent Error, No Retry	SDLC frame reject received
STATION	Retryable	BSTAT0 = I/O Exception BSTAT/ISTAT = Cyclic Redundancy Check Error

To retry errors, it is necessary to be able to reference a user-specifiable retry limit. Such limits are maintained in the network description and are referenced by the SDLC IOM error recovery procedure modules.

The first SDLC IOM module that is involved in error detection is the queue message router. When an ORE is received, BSTAT0 is tested for command complete, I/O exception, or I/O error indications. If any of these are present, the normal response module is invoked. However, if none of the preceding indications are present, the queue message router invokes #TPLIOL0, the inoperative link handler.

When a module such as #TPLXID2, #TPLCNT2, or #TPLISIN is invoked by the queue message router, BSTAT0 is checked for I/O exception or I/O error. If an exception or an error exists, #TPLIOEX, the I/O exception handler, is invoked with appropriate parameters. #TPLIOEX increments the error counter in the LKB or SOT entry, makes retry determination, restarts the OU task, and performs a read sense for I/O errors. #TPLIOEX also invokes either #TPLIOS, the inoperative station handler, or #TPLIOL0, the inoperative link handler, depending upon the error type. #TPLIOEX also invokes #TPLIOX1 to cause an error log record to be built and sent, if appropriate.

Module #TPLIOL0 sets an error code in the status field in the VMC header for any MSCP or station IOM message pending in the LKB and SOT, and then returns the messages. When appropriate, a send count is issued to the queue control table to restart the appropriate OU task, and a read sense ORE is built and sent to the appropriate queue. When the read sense ORE is returned, the queue message router invokes #TPLIOL1, the read sense response handler. If a read sense operation is not appropriate at this time, #TPLIOL1 is invoked directly by #TPLIOL0. #TPLIOL1 invokes #TPLIOX1, the error log writer, an entry point of #TPLIOEX. When control is returned to #TPLIOL1, it builds an inoperative message and sends it to the MSCP. The error code reported in this message is used in the event raised by the MSCP to inform the Control Program Facility of the error.

Control is then returned to the queue message router to await a de-activate link message, or an initialize line message from the MSCP.

Modules such as #TPLXID2 or #TPLISIN also detect errors not reported by the IOC in BSTAT, FSTAT, or ISTAT. These are errors such as invalid SDLC station address, sequence number mismatch, and invalid control field. These errors are reported to #TPLIOEX and logged, just as with the BSTAT, FSTAT, and ISTAT errors.

Read Data Store

The read data store send/receive message (SRM) is sent by the diagnostic component to the line IOM to perform the read data store function. The queue message router receives the SRM and calls #TPLRDS0. #TPLRDS0 sets read data store pending status in the LKB and calls the output scheduler. The output scheduler checks for read data store pending. If read data store pending is on, the scheduler calls #TPLRDS1. #TPLRDS1 is the highest priority function in the scheduling sequence and executes before any other pending functions. #TPLRDS1 builds the read data store ORE and sends it to the transmit OU queue. The ORE response is then handled by #TPLRDS2. #TPLRDS2 validates and error checks the BSTAT in the ORE response, performs proper cleanup, and sends the read data store SRM to the return queue.

Internal Trap

The internal trap SRM is sent by the diagnostic component to the line IOM to perform the set monitor mode function. The queue message router receives the SRM and calls #TPLTRPO. #TPLTRPO then handles the following functions:

- Trap Sync
 - A previous trap setup SRM was received with wait for sync specified.
 - Single Station
 - Set monitor mode pending is set for the station specified.
 - All Stations
 - Set monitor mode pending is set for the next station to be serviced by the scheduler.
- Trap Setup
 - The following setup functions are handled:
 - Wait For Sync (all stations)
 - The trap wait bit is set on for each configured SOT entry.
 - Wait For Sync (single station)
 - The trap wait bit is set on for the single specified station.
 - Do not Wait For Sync (no reset)
 - The trap wait and set monitor mode pending bits are set on for the next station to be serviced by the scheduler. This causes the set monitor mode I/O to perform its function immediately.
 - Do not Wait For Sync (reset)
 - The trap wait and set monitor mode pending bits are set on for the next station to be serviced by the scheduler. This causes the set monitor mode I/O to be reset immediately.

The scheduler is then called to service the appropriate station and function.

The scheduler checks for trap wait and set monitor mode pending. If both bits are on, the scheduler calls set monitor mode transmission (TPLTRP1) for the station being serviced. This is a high-priority function in the scheduling sequence and executes before all other pending functions except for the read data store function.

#TPLTRP1 sets up the transmit ORE with the Set Monitor Mode command and sends the transmit ORE to the transmit OU queue.

After receiving the set monitor mode function, the ORE is returned by the IOC to the SDLC IOM and is routed by the queue message router to the set monitor mode response (#TPLTRP2). This routine resets all set monitor mode pending and trap wait bits in all SOT entries, sets status in the trap SRM, and returns the SRM to the diagnostic component. If an error occurred, the status is set to the BSTAT returned in the ORE and proper cleanup is executed. No retry procedures are performed, regardless of the severity of the error.

DATA AREAS

Link Control Block

The LKB is in the invocation work area of the queue message router and is present when the task is active. The LKB contains pointers, the program operation block OREs, status indicators, and retry counters used for error recovery.

A pointer to the LKB is stored in register 4 and is saved throughout the task. The contents of the LKB and its relationship to other SDLC IOM areas are shown in Figure 24-7.

Machine-Wide Storage

When an activate link message is received, an area in machine-wide storage is acquired. The size of this storage is calculated from the network and controller description parameters:

- Number of stations (controller descriptions)
- Station buffer sizes

This area, shown in Figure 24-7, is used to contain the following areas:

- The service order table (SOT)
- Function operation block (FOB)
- Buffer control list
- Input buffers

SDLC (Synchronous Data Link Control) Input Areas

The synchronous data link control (SDLC) input area consists of two parts: the input buffers and the input buffer control list. This area is built in machine-wide storage by the SDLC IOM during vary-on-ND processing. Multiple input buffers are established, each large enough to contain the maximum number of frames supported by SDLC. The buffers are controlled by the input buffer control list; there is one entry for each buffer. The input buffer control list entry contains the status of the buffer, the length of the buffer, and a pointer to the buffer. Each module that needs an input buffer first checks the status in the buffer control list and uses the first buffer marked not busy. If all buffers are busy, the SDLC IOM enters the receive-not-ready state and remains in that state until an input buffer is available.

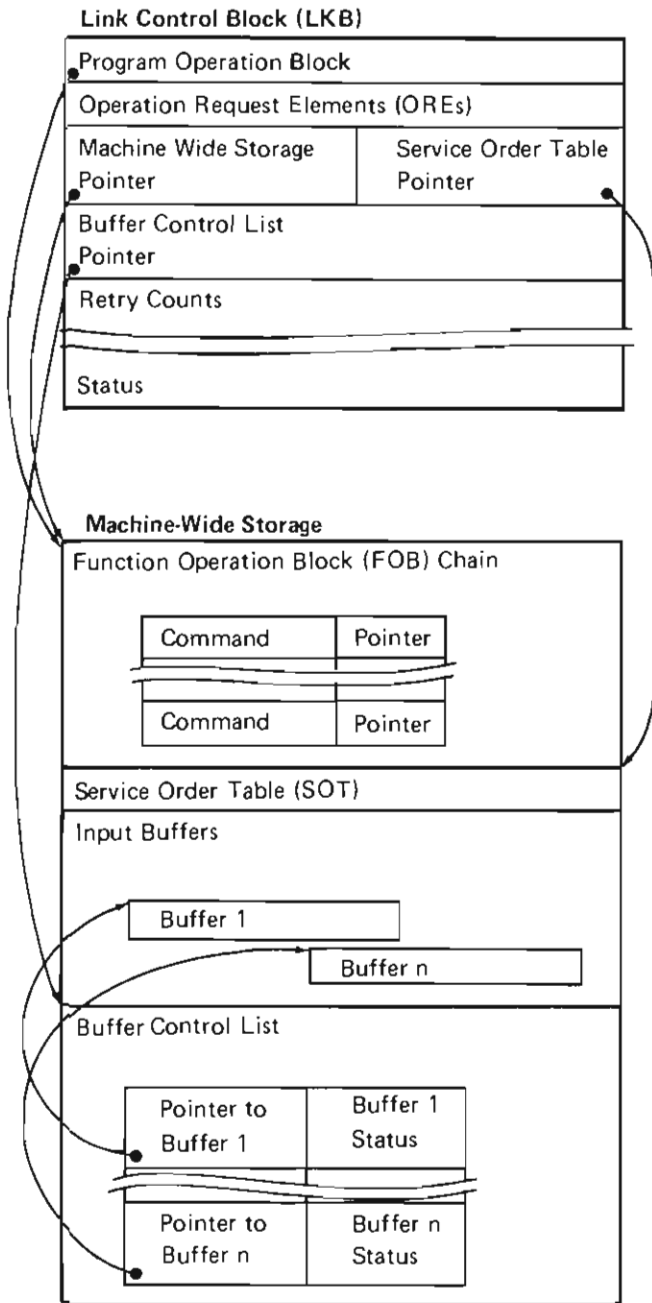


Figure 24-7. Primary SDLC IOM Areas

Service Order Table

The SOT contains an entry for each station on the line. An entry contains the following information:

- Operational status
- Pointers
- Retry counts
- Work areas

The FOBs are entries in a chain. Each entry contains an FOB command and a pointer.

The buffer control list is used to maintain the location and status of the input buffers. There is an entry in the buffer control list for each buffer. The entry contains a pointer to the buffer and the status of that buffer. A nonzero status indicates that the buffer is in use; a zero status indicates that the buffer is available for use.

STRUCTURE

The following is a list of the modules in the SDLC IOM and the function that each module performs. This list also shows how the module is invoked.

#TLXIDCK XID Format-3 Checker

Function: Validates the contents of format-3 exchange identification frame a received.

How Invoked: Within this component.

#TL1RD Request Disconnect Response Handler

Function: Notifies the station IOM or MSCP of the response received.

How Invoked: Within this component.

#TL2NAIF System Network Architecture (SNA) Interface

Function: Creates and initializes SNA control areas for SDLC to enable SNA logic processing of the first expected activate physical unit request from a boundary function.

How Invoked: Within this component.

#TL2RFMS Report Maintenance Statistics

Function: Notifies the station IOM of maintenance statistics counter overflow within SDLC secondary.

How Invoked: Within this component.

#TP2ALK0 Activate Link Processor

Function: Obtains storage and/or initializes SDLC control blocks, input buffers, buffer control list, SSCP-PU routing elements, and function operation block (FOB) list, sets line in normal disconnect mode and PU inactive, and issues set line priorities ORE to the IOC.

How Invoked: Within this component.

#TP2CINV Invalid Command Processor

Function: Establishes the command reject exception condition.

How Invoked: Within this component.

#TP2CNT Contact Processor

Function: Saves secondary station's send/receive queue address, initializes statistical data, and returns contact response to MSCP.

How Invoked: Within this component.

#TP2DCT Discontact Processor

Function: Cleans up control blocks after vary-off CD SRM has destroyed the secondary station IOM and returns discontact response to MSCP.

How Invoked: Within this component.

#TP2DM Disconnect Mode Processor

Function: Processes SDLC disconnect command received from host. Responds with proper acknowledgment and enters normal disconnect mode.

How Invoked: Within this component.

#TP2ISIN Information/Supervisory Input Frame Processor

Function: Validates received SDLC information and supervisory frames and routes for appropriate processing.

How Invoked: Within this component.

#TP2MC Mode Change Processor

Function: Process the following mode changes:

- Second Set Normal Response Mode command.
- Disconnect from host.
- Command reject condition.

How Invoked: Within this component.

#TP2OFMX Output Frame Processor

Function: Builds and sends all OREs to the transmit OU queue, then sends a read ORE to the receive OU queue.

How Invoked: Within this component.

#TP2RDS0 Read Data Store SRM Processor

Function: Handles the read data store SRM and sets pending status for subsequent execution.

How Invoked: Within this component.

#TP2RDS1 Read Data Store ORE Processor

Function: Builds and sends a read data store ORE to the transmit OU queue.

How Invoked: Within this component.

#TP2RDS2 Read Data Store SRM Responder

Function: Updates the status of the read data store SRM and returns the read data store SRM response to the return queue.

How Invoked: Within this component.

#TP2RSID Request System Service Control Point Identification Processor

Function: Builds and sends an unpaired read ORE to the receive OU queue to be ready for SDLC exchange identification processing with the host.

How Invoked: Within this component.

#TP2SDLC Secondary Line IOM Activity Controller

Function: Loops to receive an SRM or an ORE from the secondary line IOM send/receive queue and routes it for appropriate processing.

How Invoked: Other VMC components.

#TP2SNRM Set Normal Response Mode Command Processor

Function: Processes SDLC Set Normal Response Mode command.

How Invoked: Within this component.

#TP2TRP0 Set Internal Trap

Function: Processes the internal trap setup for the diagnostic component. Sets service order table status and optionally invokes scheduler to call **#TP2TRP1**.

How Invoked: Within this component.

#TP2TRP1 Send Set Monitor Mode

Function: Builds and sends the set monitor mode ORE to the transmit OU queue.

How Invoked: Within this component.

#TP2TRP2 Set Monitor Mode Response

Function: Returns the internal trap response, and handles errors and exceptions on the set monitor mode ORE.

How Invoked: Within this component.

#TP2TSTR Test Command Responder

Function: Processes the SDLC Test command to duplicate back to the host any information frame present in the Test command.

How Invoked: Within this component.

#TP2XIDR Exchange Identification Processor

Function: Processes the SDLC Exchange Identification command to prepare the response for the host.

How Invoked: Within this component.

#TPLABCN Abandon Connection

Function: Terminates a switched-line connection.

How Invoked: Within this component.

#TPLACIN Activate Connect-In

Function: Enables a line for autoanswer, manual answer, or manual dial.

How Invoked: Within this component.

#TPLACOT Abandon Connect-Out

Function: Resets a line so that the dial process is ended.

How Invoked: Within this component.

#TPLALK0 Activate Link

Function: Builds the link control block and FOB chain, reserves space for the service order table, and issues the Set Line Priority command.

How Invoked: Within this component.

#TPLALK1 Initialize Response

Function: Processes the IOC response to the Initialize command.

How Invoked: Within this component.

#TPLAUDL Autodial Response

Function: Informs the MSCP that the dial-out process is completed and a switched connection was established.

How Invoked: Within this component.

#TPLAUTO Autopoll

Function: Invokes the autopoll function.

How Invoked: Within this component.

#TPLCHEV Channel Event/Error Handler

Function: Processes channel events and errors.

How Invoked: Within this component.

#TPLCNT0 Contact Message Handler

Function: Sets the service order table entry for SNRM transmission.

How Invoked: Within this component.

#TPLCNT1 Set Normal Response Mode Command Transmission

Function: Acquires an input buffer for the receive operation, sets up the transmit and receive OREs, sets the nonsequenced Set Normal Response Mode command in the frame, and sends it to the OU queue.

How Invoked: Within this component.

#TPLCNT2 Set Normal Response Mode Command Response

Function: Validates and error checks the BSTAT, FSTAT, and control field responses obtained from the ORE. Returns the MSCP contact message as appropriate.

How Invoked: Within this component.

#TPLCOUT Connect-Out

Function: Enables the line for dialing to the station indicated in the message.

How Invoked: Within this component.

#TPLDAK0 De-activate Link

Function: Resets the IOC hardware on the line, sends a vary-off network description error to the error log queue, frees the working storage obtained at activate link time, and destroys the line IOM task.

How Invoked: Within this component.

#TPLDCIN De-activate Connect-In

Function: Resets the line so that the connect-in process is no longer active.

How Invoked: Within this component.

#TPLDCT0 Discontact Function Request Handler

Function: Sets the service order table entry for discontact transmission, and sends a vary-off controller description error to the error log queue.

How Invoked: Within this component.

#TPLDCT1 Discontact Transmission

Function: Builds the ORE and FOB for the Discontact command and transmits them to the OU queues. Sets I/O busy in the LKB and updates status counters in the LKB and SOT.

How Invoked: Within this component.

#TPLDCT2 Discontact Response Handler

Function: Checks the status of I/O completion, updates completion status, invokes inoperative station on hard errors, and returns the message to the MSCP with the status.

How Invoked: Within this component.

#TPLDELY Wait Time-Out Handler

Function: Searches the SOT for entries for delayed contact, resets the skip bit, and invokes the output scheduler.

How Invoked: Within this component.

#TPLESCR Enable Switched Connection Response

Function: Informs the MSCP that a switched connection was established.

How Invoked: Within this component.

#TPLEXCP SDLC Task Exception Handler

Function: Notifies MSCP and station IOMs of an exception occurring in the SDLC IOM.

How Invoked: Fourth level exception handler.

#TPLEXT0 Execute Test

Function: Processes the execute test message and sets pending status for subsequent execution.

How Invoked: Within this component.

#TPLFOBT FOB Time-Out Handler

Function: Processes the time-out message of the channel IOM.

How Invoked: Within this component.

#TPLINLN Initialize Line

Function: Causes the line associated with the command to be initialized with the parameters received in the network description.

How Invoked: Within this component.

#TPLIOCO Communications IOC Initialization Routine

Function: Causes the communications IOC data storage to be built and written.

How Invoked: Other VMC components.

#TPLIOC1 Write IOC Data Storage

Function: Builds the communications IOC data storage image and issues the write data storage command to the OU queue.

How Invoked: Within this component.

#TPLIOC2 Error Log During IOC Wakeup

Function: Performs error logging for errors encountered in #TPLIOC0 or #TPLIOC1.

How Invoked: Within this component.

#TPLIOEX I/O Exception Handler

Function: On a communications error, performs retry determination, saves various counts for analysis purposes, performs appropriate logging for given situations, invokes the inoperative station routine if a limit is exceeded on a station-related error, and invokes the inoperative line routine if a limit is exceeded on a line related error.

How Invoked: Within this component.

#TPLIOL0 Inoperative Link

Function: Sends response messages with permanent error status for the line and all stations on the line. Restarts the OU tasks and performs a read sense operation.

How Invoked: Within this component.

#TPLIOL1 Read Sense Response

Function: Causes an error log record with device status (DSTAT) states to be sent to the error log queue. Sends an inoperative message to the MSCP and each active station IOM to indicate the cause of the failure.

How Invoked: Within this component.

#TPLIOS Inoperative Station

Function: Clears the service order table status for the station, and builds error recovery procedures messages and sends them to the appropriate queues. The skip bit is set in the SOT to disable the station. A vary on can be issued to bring the station back on line.

How Invoked: Within this component.

#TPLISIN Information Transfer and Supervisory Frame Input

Function: Based on the frame type, the error status, and the number sent count, this module determines if an input frame is accepted as a valid I-frame that is to be sent to the station IOM or as a frame to be handled further within the SDLC primary IOM. Sequence counts are also compared to determine if any output retransmission is required.

How Invoked: Within this component.

#TPLORQH Output Request Handler

Function: Routes output request messages based on destination station status in the SOT entry.

How Invoked: Within this component.

#TPLPEVT Post Event Handler

Function: Processes post error event messages from the channel IOM.

How Invoked: Within this component.

#TPLQRT0 Queue Message Router

Function: Analyzes send/receive queue elements (OREs and messages) and routes them to the appropriate module for processing.

How Invoked: Other VMC components.

#TPLQRT1 Queue Message Router (Nonmainline)

Function: Analyzes send/receive queue elements (OREs and messages) not routed directly by #TPLQRT0 and routes them to the appropriate module for processing.

How Invoked: Within this component.

#TPLRSTR Reset Response

Function: Sets status and returns the response message.

How Invoked: Within this component.

#TPLSCED Output Scheduler

Function: Selects an eligible station and starts an output operation. Prepares the ORE, program operation block, FOB, and I-frame for output request messages. Prepares ORE, FOB, and S-frame for the supervisory response frame. Builds a receive ORE/FOB. Performs a send messages operation to send the messages to the transmit and receive OU queues.

How Invoked: Within this component.

#TPLSLPR Set Line Priorities Response

Function: Processes the IOC set line priorities response.

How Invoked: Within this component.

#TPLSRMR Response Handler

Function: Performs input buffer and message management for the response message.

How Invoked: Within this component.

#TPLRDS0 Read Data Store Message Handler

Function: Handles the read data store SRM and sets pending status in the link control block for subsequent processing.

How Invoked: Within this component.

#TPLRDS1 Read Data Store Execution

Function: Sets up the transmit ORE and sends it to the transmit OU queue.

How Invoked: Within this component.

#TPLRDS2 Read Data Store Response Handler

Function: Validates and error checks the BSTAT in the ORE response, performs cleanup, and sends the read data store SRM to the response queue.

How Invoked: Within this component.

#TPLTRP0 Internal Trap Setup

Function: Processes the trap setup and diagnostic component. Sets SOT status for affected entries, then calls the scheduler to cause the set monitor mode ORE to be sent.

How Invoked: Within this component.

#TPLTRP1 Set Monitor Mode Transmission

Function: Builds the transmit ORE, sets the Set Monitor Mode command in the ORE, and sends it to the transmit OU queue.

How Invoked: Within this component.

#TPLTRP2 Set Monitor Mode Response

Function: Posts back the trap SRM and handles the errors and exceptions on the set monitor mode ORE.

How Invoked: Within this component.

#TPLTST1 SDLC Test Transmission

Function: Builds the transmit and receive OREs, builds the nonsequenced Test command in the frame, and sends the preceding to the OU queues.

How Invoked: Within this component.

#TPLTST2 Test Response

Function: Returns the execute test message with the appropriate status.

How Invoked: Within this component.

#TPLWDIS Write Data I-Frame Response

Function: Processes the write data ORE for information commands transmitted. Performs error checking and initiates error recovery when possible.

How Invoked: Within this component.

#TPLWDNS Write Data ORE
Handler-Nonsequenced Sent

Function: Processes the write data ORE when the ORE key indicates that a nonsequenced command or S-frame was transmitted. Initiates error recovery if a recoverable error is indicated.

How Invoked: Within this component.

#TPLXID0 Request Exchange Identification
Handler

Function: Sets the service order table entry for exchange identification transmission and returns the MSCP message.

How Invoked: Within this component.

#TPLXID1 Exchange Identification Transmission

Function: Acquires an input buffer for the receive operation, builds the transmit and receive OREs, builds the nonsequenced exchange identification in the frame, and sends the preceding to the OU queues.

How Invoked: Within this component.

#TPLXID2 Exchange Identification Response

Function: Validates and error-checks the BSTAT, FSTAT, and control field responses obtained from the ORE. Builds a request contact message if appropriate

How Invoked: Within this component.



Local I/O Manager

INTRODUCTION

The local I/O manager (IOM) provides the control of certain locally attached I/O devices (printers, card readers, diskette, console, and tape devices). Local IOMs are device dependent and provide the interface between the System/38 instructions and the internal microprogram instructions. There is a local IOM task for each online device and there is a set of IOM routines for each device type supported. Multiple IOM tasks for similar devices can use the same IOM routines.

An overview to local IOM processing is shown in Figure 25-1. A local IOM task, the associated operational unit (OU) task, the related queues (including the IOM queue and the OU queue), and related tables are constructed by the machine services control point (MSCP) as a result of Modify Logical Unit Description (vary-on) instruction and are destroyed at vary-off time. Although the IOM task is created by the MSCP, the IOM task destroys itself at vary-off time.

A local IOM task is dispatched (placed on the prime task dispatching queue) by the MSCP. A local IOM task then performs additional initialization before waiting on the IOM queue associated with this task. Once an IOM task is waiting on the IOM queue, it is ready to receive additional modify logical unit description requests; I/O requests are then passed from the issuing process to the IOM queue. Modify logical unit description and I/O requests requiring actions by the adapter are formatted into operation request elements (OREs) and are sent by the local IOM task to the OU queue. The OU task returns the completed OREs to the IOM queue. The IOM task always waits on the associated IOM queue to receive additional and completed requests. When a machine interface request is completed, the request is returned to the machine interface response queue designated for the process issuing the Modify Logical Unit Description or Request I/O instruction. There is a local IOM for each of the following devices:

- Printer
- Diskette
- Multi-function card unit (MFCU)
- 3410/3411 Magnetic Tape
- 3430 Magnetic Tape
- Console

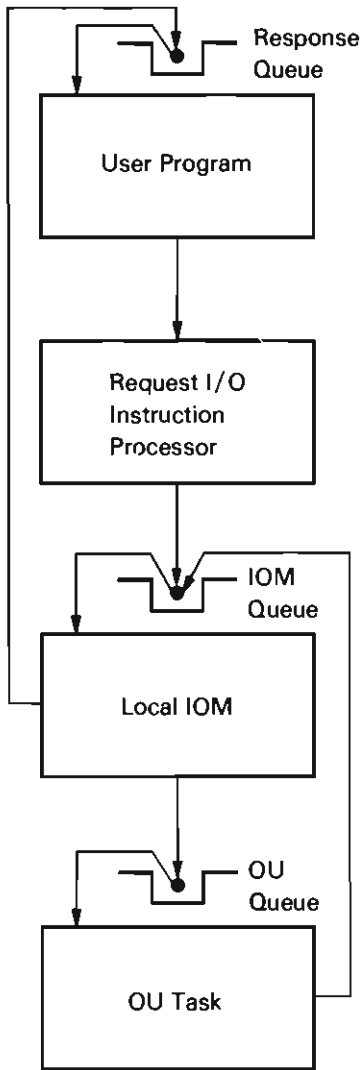


Figure 25-1. Local IOM Overview

The basic flow of a local IOM is as follows:

1. Create and initialize fields (modify logical unit description—vary-on):
 - Get machine-wide storage for all OREs (operation request elements).
 - Build keys and pointers.
 - Build the program operation block and messages for all I/O operations.
 - Begin never-ending loop to receive messages from the IOM queue.
2. Receive a request (an I/O request, a modify request, or a completed OU request).
3. Decode the request:
 - Check the request by key or function to invoke the appropriate routine to process the request.
 - If the request is invalid, return the request with invalid status indicated.
4. Return to step 2.

There are local IOM routines that process each possible major message type (vary on, vary off, and so on) as shown in Figure 25-2. The routines interface with the OU task, the user, or both. All local IOMs are similar in structure. A local IOM consists of a main driver routine that invokes the different functions to be performed. The function invoked can either be in the same module as the main-driver module, or in a different module. The appropriate module to be invoked is determined from the key and function fields of the request message. The operations of the local IOM functions are described in the following paragraphs. The IOM for the diskette is used for this description because it is representative of the other local IOMs.

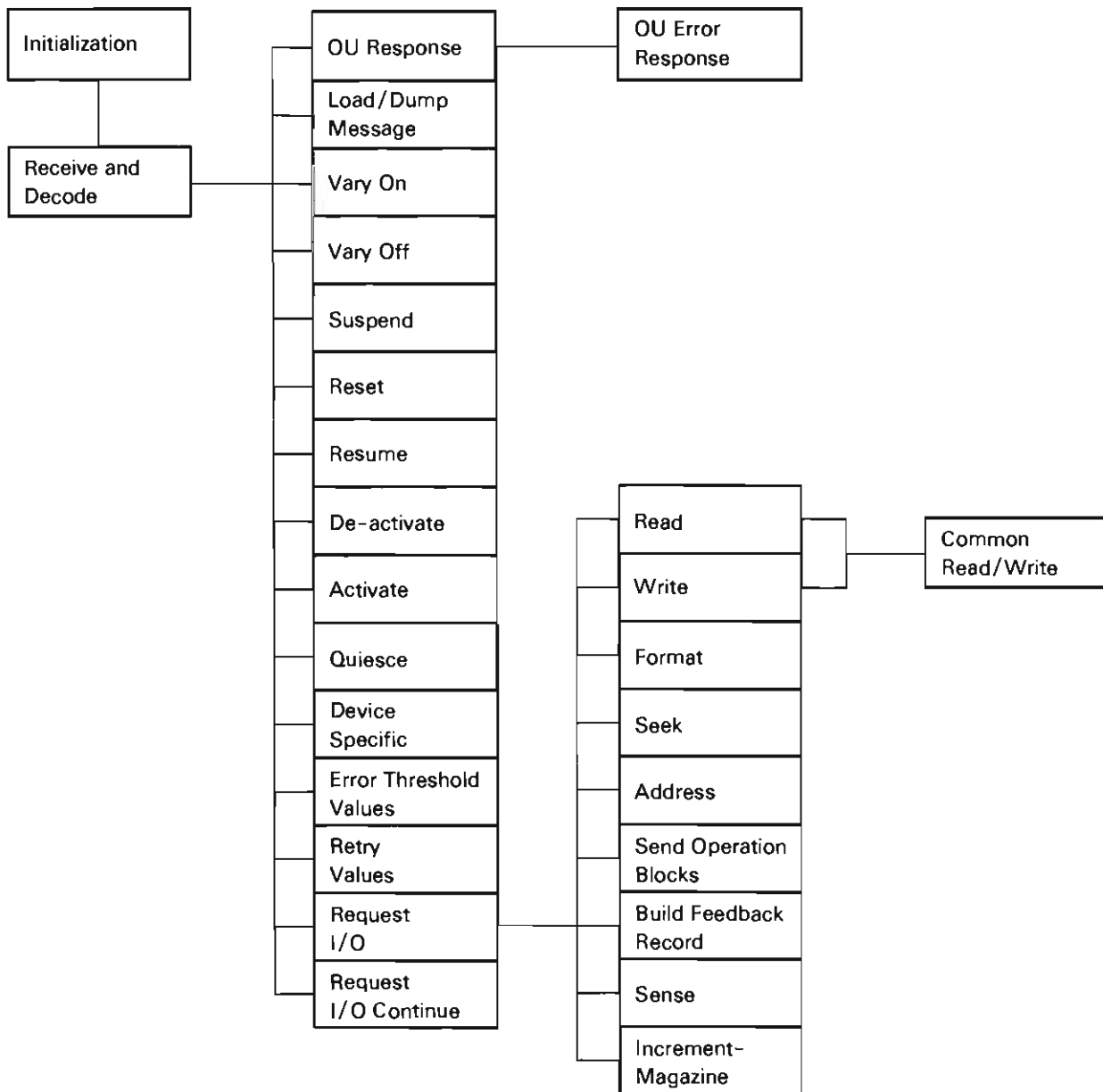


Figure 25-2. Local IOM Functions

Initialization: This function obtains space for the ORE and obtains addressability to all objects used. Initialization then builds the message and program operation block used for all operations and builds the queue and message area used by load/dump.

Receive and Decode: This function, performed by the main-driver module, dequeues a message from the IOM queue and validates the key or function field. If the key or function field is valid, this function invokes the proper routine to process the request. If the key or function is not valid, this routine indicates an invalid status and returns the message to the issuing process.

OU Response: This function validates the response by checking the basic status (BSTAT) that was returned from the device, and if invalid, invokes the error processing routine. If valid, the OU response routine updates fields based on flag bytes, and performs a compare descriptor operation if a load/dump read descriptor command is specified. The OU response function updates the current diskette address (cylinder-head-sector). Then, depending on the command specified, the request I/O routine (if more work is to be performed), the feedback record routine (if the work is completed), or the load/dump function is invoked.

Load/Dump Message: This function is available only for a magnetic media device. It builds parameters for a load or dump function, and then processes the request.

Vary On: This function establishes addressability to the required data areas and initiates a startup sequence for the device. This function then responds to the request.

Vary Off: This function releases the storage buffers and issues a check-reset to the device. Vary off then responds to the request and causes the IOM task itself to be destroyed.

Suspend: This function completes processing on the request I/O in process, then raises the key to a value that inhibits additional processing of request I/O messages and responds to the request.


Reset: This function halts the OU task, dequeues all OREs from the OU queue, and then, if required, restarts the OU task. Reset then returns the current message and all messages that are currently enqueued on the IOM queue. Reset then releases any storage buffers as required, and responds to the request.

Resume: This function enables message processing and then responds to the request.

De-activate: This function unloads the diskette (if one is loaded), then positions the diskette magazine in the home position (manual slot 1) and logs the statistical information, and then responds to the request.

Activate: This function creates a load/dump task if required, obtains the required device specific parameters, and then responds to the request.

Quiesce: This function processes all request I/O messages on the IOM queue, raises the key so that no additional request I/O messages will be processed, and then responds to the request.




Device Specific: This function obtains the required device specific information, and for some devices, performs device related functions. This processing includes building the required OREs and sending them to the OU tasks and waiting for the OU task response.

Error Threshold Values: This function sets the threshold values for logging read and write data check errors.

Retry Values: This function sets the retry values for retrying read and write data check errors.

Request I/O: This function first validates the message. The appropriate routines are then invoked to process each request descriptor (RD). The required OREs for each RD are built and sent to the OU task. When all RDs have been processed, the request I/O message is returned to the user process.

Request I/O Continue: This function resets all error flags and lowers the key value so that request I/O messages can be processed.



Internal Cleanup Routine

Local IOMs provide recovery from damaged objects. An internal cleanup routine is added to each of the local IOMs when an internal microprogram instruction exception is detected. This routine first determines the type of exception. If the exception is a source/sink data exception, the cleanup routine handles the exception. If the exception is something other than a source/sink data exception, the cleanup routine calls a common exception cleanup routine.

DATA AREAS

The data areas for the local IOM are described under *Source/Sink Data Areas* in the *Vertical Microcode Overview* section of this publication.

STRUCTURE

The following is a list of the modules in the local IOMs and the function that each module performs. This list also shows how the module is invoked.

Console Local IOM

#LOCNIOM Console IOM

Function: Provides the interface between the user and the OU task that performs the console function. This is a queued interface, where this module accepts requests on its input queue, performs the task, and then responds to the request by returning it to the appropriate queue. This module contains the following procedures:

INITIALZ	Initialization
OURSPCHK	Process response from OU task
OUSTATCK	BSTAT analysis
DSTATCHK	Device status (DSTAT) analysis
CHNLERR	Channel error
REQIO	Process request I/O message
RQREAD	Read without control
RQCANCEL	Cancel a read modified
RQWRITE	Write with or without control
RQCNTL	Control
VARYON	Vary on

VARYOFF	Vary off
ACTSESS	Activate logical session
DACTSESS	De-activate logical session
QUIESCE	Quiesce
RESET	Reset
SUSPEND	Suspend
RESUME	Resume
CONTINUE	Continue
CHNLERR	Channel error message
CHNLPEVT	Post event message
BUILDFBR	Build feedback record
BUILDERL	Error logging
ELOGRESP	Error logging response
RETRYCMD	Retry failing command
SENDLEGM	Unrecoverable I/O light-emitting diode message to system control adapter (SCA) IOM
DUALERP	Perform recovery for multiple commands
LEDRMGB	Process light-emitting diode message from SCA IOM
GATEOPNA	Perform input/output controller (IOC) gate open (any) macro
GATEOPN	Perform IOC gate open macro
GATECLZE	Perform IOC gate close (C) macro
TIMEHNDL	Handle 2-second-interval timer message and issue system request read modified
CANRDMOD	Cancel system request read modified
OURSTART	Performs SSRSTOU macro
OURSET	Resets IOC console
REWRITE	Writes last saved screen
OUFAIL	Process OU task failure message
FOBTIMOT	Function operation block (FOB) time-out
PREPAOC	Preprocess request I/Os
VERIFY	Verify request description
KBRDCNTL	Update keyboard state
MOVEDATA	Move data from IOM buffer to users buffer
CLEANUP	Reset error recovery procedure parameters
STDERP	Standard ERP
HARDERR	Set hardware error

How Invoked: Within the console IOM.

#LOMACNM Diskette IOM Channel Error Recovery Module

Function: Contains routines to process errors encountered by the channel. Determines the error type and the action to be performed. This module contains the following procedure:

LOMACHEM	Process channel messages
LOMAC101	Channel error
LOMAC204	Channel time-out on an Autoload command
LOMAC208	Channel time-out
LOMAC504	Read sense error during channel error recovery
LOMACPOS	Position heads

How Invoked: Within the diskette IOM.

Diskette Magazine Drive Local IOM

#LOMACOM Diskette IOM Common Function Processor

Function: Contains the routines that perform common functions. This module contains the following procedures:

LOMASEND	Send the OREs to the OU queue
LOMACFOB	Complement fields in FOBs
LOMASPCK	Check space available on current cylinder
LOMAALEB	Build the address list elements for read/write
LOMAPERF	Performance analyzer
LOMALADD	Update last address
LOMACBLF	Build load/dump feedback record
LOMACBRF	Build REQIO feedback record
LOMABLOG	Build the error log message
LOMACNVT	EBCDIC/ASCII conversion
LOMACUPN	Unpin storage

How Invoked: Within the diskette IOM.

#LOMAERP Diskette IOM Error Recovery Procedure

Function: Contains routines to process errors encountered by the OU task. Determines the error type and the action to be performed. This module contains the following procedures:

LOMAOUE	OU error response
LOMANOUE	OU error response (no error entry)
LOMAEIOE	I/O error
LOMAAUTE	Decode autoloader error
LOMAMEDE	Decode media error
LOMA0100	Operation program error
LOMA0103	Overrun
LOMA0104	Disconnect
LOMA0105	Parity-1 error
LOMA0106	Parity-2 error
LOMA0107	Command reject
LOMA0108	Control address mark found
LOMA0201	Wrap
LOMA0202	Autoloader parity error
LOMA0203	Invalid autoloader function code
LOMA0205	Erase current error
LOMA0206	Autoloader write/erase current error
LOMA0207	Write gate
LOMA0301	Autoloader carriage bed malfunction
LOMA0302	Autoloader picker malfunction/fail to eject
LOMA0303	Autoloader fail to pick
LOMA0304	Autoloader diskette window fail to open
LOMA0305	Autoloader device cover open
LOMA0306	Autoloader not oriented
LOMA0307	Autoloader out of sequence
LOMA0308	Speed check
LOMA0309	CHN mismatch
LOMA0403	Data CRC error on read
LOMA0404	Data CRC error on write and read verify
LOMA0405	ID CRC error on read
LOMA0406	ID CRC error on write

LOMA0407	Not oriented-1
LOMA0408	Not oriented-2
LOMA040A	No record found
LOMA0505	Invalid BSTAT/DSTAT
LOMA0506	Read sense error
LOMA0507	Autoloader parity error on read sense
LOMA0508	Retry stack limit exceeded
LOMA050C	Failed to read hex FF in ID field
LOMA9998	Channel busy
LOMAEVFY	Verify and log error
LOMAEUCT	Update retry and error SAR counters
LOMAEISK	Insert on retry list
LOMAEMED	Media error
LOMAESEK	Seek/compare error
LOMAESKN	Seek/no compare error
LOMAEFMT	Format diskette error
LOMAEFID	Format read ID error
LOMAECC	Single bit data CRC error correction
LOMAEFBR	Build feedback record

How Invoked: Within the diskette IOM.

#LOMAIOM Diskette IOM Main Module

Function: Provides the interface between the user and the OU task that performs the diskette function. It is a queued interface, where this module accepts requests on its input queue, performs the task, and responds to the request by returning the request to the appropriate queue.

This module decodes the messages and invokes one of the following functions:

- MSCP/Modify Logical Unit Description instruction processor (#LOMALUD)
- Request I/O processor (#LOMARIO)
- Load/dump processor (#LOMALDM)
- OU response processor (#LOMAOUM)
- Channel error processor (#LOMACNM)

This module contains the following procedures:

LOMAICLN Cleanup for IMPL exceptions
LOMAINIT IOM initialization

How Invoked: By MSCP at vary-on-LUD time.

#LOMALDM Diskette IOM Load/Dump Processor

Function: Processes messages that are sent to the diskette IOM as a result of a load/dump request. This module contains the following procedures:

LOMALDCM Handle load/dump continue requests
LOMALDNM Handle load/dump normal requests
LOMALDWB Handle load/dump write block requests
LOMALDRB Handle load/dump read block requests
LOMALDSB Handle load/dump space block requests

How Invoked: Within the diskette IOM.

#LOMAOUM Diskette IOM OU Response Module

Function: Processes the OU response. This module contains the following procedures:

LOMAOUR Handle OU response
LOMAOFMT Handle OU response for Format RD command
LOMAOVTC Handle OU response for Read VTOC command
LOMAOINC Handle OU response for Increment command
LOMAOSEK Handle OU response for Seek command
LOMAOULD Handle OU response for load/dump descriptor request
LOMAOSDR Increment SDR counters

How Invoked: Within the diskette IOM.

#LOMALUD Diskette IOM MSCP/Modify Logical Unit Description Instruction Processor

Function: Processes messages that are sent to the diskette mainline as a result of a modify logical unit description request. This module contains the following procedures:

LOMAVONN	MSCP vary-on request
LOMAGALE	Get address list element space
LOMAVOFF	MSCP vary-off request
LOMAFALE	Free address list element space
LOMAACTS	MSCP activate session request
LOMADSAC	Modify Logical Unit Description instruction device-specific area change request
LOMACCVH	Process volume and header information
LOMADACS	MSCP de-activate session request
LOMAQUIS	Modify Logical Unit Description instruction quiesce session request
LOMASUSP	Modify Logical Unit Description instruction suspend session request
LOMARESM	Modify Logical Unit Description instruction resume session request
LOMARSET	Modify Logical Unit Description instruction reset session request
LOMALDMC	Modify Logical Unit Description instruction load/dump mode change request
LOMARVSC	Modify Logical Unit Description instruction retry value sets change request
LOMAETSC	Modify Logical Unit Description instruction error threshold sets change request
LOMASMSG	Send response to Modify Logical Unit Description instruction/MSCP

How Invoked: Within the diskette IOM.

#LOMARIO Diskette Request I/O Processor

Function: Processes messages that are sent to the diskette IOM as a result of a request I/O. This module contains the following procedures:

LOMARCON	Request I/O continue
LOMARIOR	Request I/O message router
LOMARLAD	Process Address command
LOMARINC	Process Increment command
LOMARSEK	Process Seek command
LOMARHLT	Process Halt command
LOMARWRN	Process Request I/O Write Request Descriptor command
LOMARRDN	Process Request I/O Read Request Descriptor command
LOMARWV	Read/write common function
LOMAWVTC	Handle write of VTOC
LOMARDAT	Process Request I/O Read command
LOMAWDAT	Process Request I/O Write command
LOMAWVT2	Handle write of VTOC (part 2)
LOMAWVT3	Handle write of VTOC (part 3)
LOMARVTC	Process Read VTOC command
LOMAWVIP	Process Write Initial Program Load (IPL) Sectors command
LOMARVIP	Process Read IPL Sectors command
LOMARFMT	Process Format command

How Invoked: Within the diskette IOM.

#LOMATOM Time-out Message Processor

Function: Process time-out messages sent to the 72MD IOM as the result of a modify LUD request. This module contains the following procedure:

LOMATOMH	Modify Logical Unit Description time-out message handler
----------	--

How Invoked: Within the diskette IOM.

MFCU Local IOM

#LOMFIOM MFCU IOM Mainline

Function: Provides the interface between the user procedure and the OU task performing the MFCU function. This is a queued interface where this module accepts requests on its input queue, performs the task, and then responds to the request by returning the request to the appropriate queue. This module contains the following procedures:

#LOMFVON	Handle vary-on
#LOMFVOF	Handle vary-off
#LOMFACT	Handle activate session
#LOMFDAC	Handle de-activate session
#LOMFQSC	Handle quiesce
#LOMFRST	Handle reset
#LOMFRES	Handle resume
#LOMFSSP	Handle suspend
#LOMFRCN	Handle request I/O continue
#LOMFRIO	Handle request I/O

#LOMFSFC	Process Single Function command
#LOMFPPT	Process Punch and Print command
#LOMFRPX	Process Read and Punch or Read and Print command
#LOMFRPP	Process Read and Punch and Print command
#LOMFOUR	Handle OU task response
#LOMFNRS	Handle normal OU response
#LOMFERS	Handle error OU response
#LOMFRTN	Return VMC message
#LOMFDSA	Process the device-specific area change request
#LOMFCER	Handle channel error messages
LOMFCLEN	Handle error recovery cleanup
LOMFRSOU	Restart OU task
LOMFELOG	Build error log message
BLDFBRE	Build feedback record
LOMFX96	Translate 96 characters
LOMFX128	Translate 128 characters
LOMFSNDR	Send ORE to OU
LOMFRDSN	Build and send read sense command to OU
LOMFRDDS	Build and Send Data Store command
LOMFCEWU	Wrap-up channel error processing
#LOMFWTR	Build wait for ready ORE
LOMFECOD	Determine error code for error log message
LOMFICLN	MFCU exception handler

How Invoked: Within the MFCU IOM.

3410/3411 Magnetic Tape Local IOM

#LOTACOM 3410/3411 Magnetic Tape IOM
Common Function Processor

Function: This module consists of routines that perform common functions for the tape IOM. The module contains the following procedures:

LOTAOUR	Handle OU response
LOTAGFMS	Get/free machine-wide storage
LOTABEFR	Build and send feedback record (error response)
LOTACNVT	Data conversion
LOTAELOG	Error log message
LOTAGVST	Generate volume statistics
LOTATRJB	Terminate all jobs
LOTARSOU	Restart OU task
LOTARMVE	Remove OU message
LOTAQMSG	Requeue request I/O message
LOTARFRM	Return first request I/O message
LOTATPST	Examine tape status
LOTAVSDR	Communicate with volume statistical data record manager
LOTACBER	Build channel busy error log message
LOTASLOG	Send error log message
LOTANBTS	Recalculate number of blocks to skip

How Invoked: Within the 3410/3411 Magnetic Tape IOM.

#LOTAERP 3410/3411 Magnetic Tape IOM Error Recovery Procedure

Function: This module is called when the response from the OU task indicates that an error condition has occurred. It determines the error type and the action to be performed. This module contains the following procedures:

LOTAQUE	Handle OU error response
LOTAEDCD	Error decoder
LOTAEIOE	I/O error handler-1
LOTAEIO1	I/O error handler-2
LOTARSID	Determine residual operation count
LOTAERDC	Read data check error recovery
LOTAEWDC	Write data check error recovery
LOTAEROR	Handle read overrun error recovery
LOTAEWOR	Handle write overrun error recovery
LOTAEIOX	Handle I/O exception
LOTAEWLR	Handle wrong length record retrying
LOTAECDR	Handle command rejects
LOTAETNA	Handle tape not available error recovery
LOTAESVC	Handle start velocity error recovery (error)
LOTAEIDB	Handle phase-encoded identification burst error recovery (error)
LOTAERDN	Read noise error recovery (error)
LOTAERBD	Read back data loop write-to-read
LOTAEWUP	Error wrap-up
LOTAEECD	Erase gap data check error recovery (error)
LOTAEICD	Illegal command error recovery (error)
LOTAEW CZ	Word count zero error recovery (error)
LOTAEUST	Update error statistics
LOTARDEC	Determine read data check error code
LOTAWTEC	Determine write data check error code
LOTAECHN	Channel error recovery
LOTAEDNA	Drive not attached error recovery

How Invoked: Within the 3410/3411 Magnetic Tape IOM.

#LOTAIOM 3410/3411 Magnetic Tape IOM
Mainline

Function: This module provides the interface between the user and the OU task that performs the tape function. This is a queued interface where this module accepts requests on its input queue. Control type requests are processed immediately upon receipt. Request I/O and load/dump requests are dequeued from the main input queue and enqueued on separate device queues. These separate device queues are then serviced in a sequential manner to prevent one device from monopolizing the tape control unit. Responses to requests are accomplished by returning the request to the appropriate request queue.

This module decodes the message and invokes one of the following functions:

- MSCP/MODLUD processor (#LOTALUD)
- Request I/O processor (#LOTASCH)
- Common function processor (#LOTACOM)
- Error recovery processor (#LOTAERP)

This module contains the following procedures:

LOTAROUT	IOM message router
LOTAINIT	Initialization routine
LOTAICLN	Tape exception handler

How Invoked: Other VMC components.

#LOTALUD 3410/3411 Magnetic Tape IOM
MSCP/Modify Controller Description
(CD)/Modify LUD Processor

Function: Processes messages that are sent to the tape IOM as a result of a modify controller description or a modify logical unit description request. This module contains the following procedures:

LOTAVNCD	MSCP vary-on CD request
LOTAVFCD	MSCP vary-off CD request
LOTAVNLU	MSCP vary-on LUD request
LOTAVFLU	MSCP vary-off LUD request
LOTAActs	MSCP activate session request
LOTADSAC	Modify LUD device-specific area change request
LOTADACS	MSCP de-activate session request
LOTAQUIS	Modify LUD quiesce session request
LOTASUSP	Modify LUD suspend session request
LOTARESM	Modify LUD resume session request
LOTARSET	Modify LUD reset session request
LOTALDMC	Modify LUD load/dump mode change request
LOTARVSC	Modify LUD retry value sets change request
LOTAETSC	Modify LUD error threshold sets change request
LOTASDOB	Send OREs to the OU queue and wait for response
LOTASMSG	Send response to Modify LUD or MSCP queue
LOTASZSH	Seize the object
LOTARLSE	Release the object
LOTAGALE	Get storage for address list element
LOTAFAL	Free storage for address list element
LOTAMLTD	Handle second time-out message
LOTARMRM	Return Modify LUD reset and load/dump abort message

How Invoked: Within the 3410/3411 Magnetic Tape IOM.

#LOTASCH 3410/3411 Magnetic Tape IOM Scheduler

Function: Processes messages that are sent to the tape IOM as a result of a request I/O or load/dump request. This module contains the following procedures:

LOTARCON	Request I/O and load/dump continue
LOTASCHD	Scheduler
LOTAALB	Build address list element
LOTAPIN	Pin/unpin storage
LOTAPSTE	Handle channel post event message
LOTASEND	Send ORE to OU task
LOTARIOR	Request I/O router
LOTARDWR	Request I/O read and write
LOTASRCL	Request I/O space rewind, write tape marks, and clear
LOTALDDP	Load/dump message router
LOTALDTP	Request I/O Load Tape command
LOTACKTP	Request I/O Check Tape command
LOTALTRS	Load tape read sense
LOTADQMW	Dequeue request I/O and load/dump messages
LOTALDWB	Handle load/dump write block request
LOTALDRB	Handle load/dump read block request
LOTALDSB	Handle load/dump space block request
LOTALDCD	Handle load/dump compress data request
LOTALDDD	Handle load/dump decompress data request

How Invoked: Within the 3410/3411 Magnetic Tape IOM.

3430 Local IOM

#LOGSIOM 3430 IOM Mainline Processor

Function: Provides the interface between the user and the OU task that performs the 3430 tape function. This is a queued interface where this module accepts requests on its input queue. Control type requests are processed immediately upon receipt. Request I/O and load/dump requests are dequeued from the main input queue and enqueued on separate device queues. These separate device queues are then serviced in a sequential manner to prevent one device from monopolizing the tape control unit. Responses to requests are accomplished by returning the request to the appropriate request queue.

This module also decodes the message and invokes one of the following functions:

- Time-out message processor (#LOGSTOM)
- Channel error message processor (#LOGCEM)
- OU response message processor (#LOGGOU)
- Channel attention message processor (#LOGCAM)
- Modify CD/Modify LUD message processor (#LOGMLM)
- Request I/O message processor (#LOGRIM)
- Load/dump message processor (#LOGLDM)
- Error recovery processor (#LOGERP)
- Common function processor (#LOGCOM)

This module contains the following procedures:

LOGSIOIT	Initialization routine
LOGSIORM	Receive message handler
LOGSIOMR	IOM message router
LOGSIODS	Device scheduler
LOGSIODT	Destroy task routine
LOGSIOCU	IOM cleanup routine

How Invoked: Other VMC component.

#LOGSTOM Time-out Message Processor

Function: Processes time-out messages sent to the 3430 tape IOM as a result of a modify LUD request. This module contains the following procedure:

LOGSTOMM Modify LUD time-out message handler

How Invoked: Within the 3430 tape IOM.

#LOGGCEM Channel Error Message Processor

Function: Processes channel error messages that have occurred as a result of a channel error. This module contains the following procedure:

LOGGCERM Channel error handler

How Invoked: Within the 3430 tape IOM.

#LOGGOUM 3430 Tape IOM OU Response Processor

Function: Processes the OU response for messages sent to the OU task.

How Invoked: Within the 3430 tape IOM.

#LOGGCAM Channel Attention Message Processor

Function: Processes the channel attention messages sent to the 3430 tape IOM. This module contains the following procedure:

LOGSCAPE Handle post event messages

How Invoked: Within the 3430 tape IOM.

#LOGGMLM 3430 Tape IOM Modify Controller Description and Modify Logical Unit Description Processor

Function: Processes messages that are sent to the 3430 tape IOM as a result of a modify controller description request and a modify logical unit description request. This module contains the following procedures:

LOGGMVNC Handle modify CD vary-on request
LOGGMVNL Handle modify lud vary-on request
LOGGMACT Handle modify LUD device specific area change request
LOGGMLDX Handle modify LUD load/dump exchange mode request
LOGGMERV Handle modify LUD error retry values change request
LOGGMETV Handle modify LUD error threshold values change request
LOGGMQIS Handle modify LUD quiesce session request
LOGGMSSP Handle modify LUD suspend session request
LOGGMRST Handle modify LUD reset session request
LOGGMRSM Handle modify LUD resume session request
LOGGMDAC Handle modify LUD deactivate session request
LOGGMVFL Handle modify LUD vary-off LUD request
LOGGMVFC Handle modify LUD vary-off CD request
LOGGMRMG Send response to modify CD/modify LUD message

How Invoked: Within the 3430 tape IOM.

#LOSGRIM Request I/O Processor

Function: Processes messages that are sent to the 3430 tape IOM as a result of a request I/O request. This module contains the following procedures:

LOSGRICM	Handle request I/O continue request
LOSGRINM	Handle request I/O normal request
LOSGRIBW	Handle write block request
LOSGRIRB	Handle read block request
LOSGRICT	Handle check tape request
LOSGRIST	Handle space block and space file requests
LOSGRIRT	Handle rewind and rewind/unload requests
LOSGRJET	Handle erase tape request
LOSGRJWT	Handle write tape mark request
LOSGRIVV	Validate Request I/O request and request descriptors

How Invoked: Within the 3430 tape IOM.

#LOSGLDM Load/Dump Message Processor

Function: Processes load/dump messages received by the 3430 tape IOM. This module contains the following procedures:

LOSGLDCM	Handle load/dump continue request
LOSGLDNM	Handle load/dump normal request
LOSGLDWB	Handle load/dump write block request
LOSGLDRB	Handle load/dump read block request
LOSGLDSB	Handle load/dump space block request
LOSGLDCD	Handle load/dump compress data request
LOSGLDD	Handle load/dump decompress data request

How Invoked: Within the 3430 tape IOM.

#LOGSERP**Error Recovery Processor**

Function: Handles the error recovery processing when the OU response from the OU task indicates an error condition. This module contains the following procedures:

LOGS3000	Handle error processing for select in error, power off or disable	LOGS5300	Handle error processing for command reject (write to file protected)
LOGS3200	Handle error processing for disconnect inactive	LOGS5400	Handle error processing for 3430 detected overrun
LOGS3300	Handle error processing for interface check	LOGS5500	Handle error processing for backward at BOT
LOGS3400	Handle error processing for adapter detected BUS IN check	LOGS5700	Handle error processing for data check (Read command)
LOGS3500	Handle error processing for adapter detected BUS OUT check	LOGS5800	Handle error processing for data check (Read command)—with LWR failure
LOGS3600	Handle error processing for unexpected subsystem status/CDADDR	LOGS5900	Handle error processing for data check (Write command)
LOGS4000	Handle error processing for unexpected channel response register contents	LOGS5A00	Handle error processing for data check (Write command)—with LWR failure
LOGS4100	Handle error processing for System/38 channel overrun	LOGS5D00	Handle error processing for not capable (not PE and not GCR)
LOGS5000	Handle error processing for equipment check	LOGS5E00	Handle error processing for ID burst check
LOGS5100	Handle error processing for 3430 detected BUS OUT check	LOGS5F00	Handle error processing for word count zero
LOGS5200	Handle error processing for intervention required	LOGS6000	Handle error processing for unit check (DSE command)
		LOGS6100	Handle error processing for unit check
		LOGS7000	Handle error processing for invalid BSTAT/DSTAT
		LOGS7100	Handle error processing for read sense failure
		LOGS7200	Handle error processing for retry stack overflow
		LOGS7800	Handle error processing for IOM error
		LOGS9800	Handle error processing for operation program error
		LOGS9998	Handle error processing for operation program error (channel busy)
		LOGSGEUOC	Update request I/O RD or load/dump message and prepare failing OU message for retrying
		LOGSERFR	Return feedback record to load/dump or request I/O queue

How Invoked: Within the 3430 tape IOM.

#LOGSCOM 3430 Tape Common Function
Processor

Function: This module consists of routines that perform common functions for the 3430 tape IOM. The module contains the following procedures:

LOGSCBAL	Build address list elements
LOGSCBLF	Build load/dump feedback record
LOGSCBRF	Build and send request I/O feedback record
LOGSCCNV	Convert EBCDIC data to ASCII for output and convert ASCII data to EBCDIC for input
LOGSCELG	Generate and send error log message
LOGSCFLU	Flush IOM and device queues
LOGSCPIN	Pin/unpin storage
LOGSCRMV	Remove OU message from OU task queue
LOGSCSND	Set up and send OU message to OU task queue
LOGSCTRJ	Terminate all jobs

How Invoked: Within the 3430 tape IOM.

3262/5211 Printer Local IOM

#LOTZERP 3262/5211 Printer Error Recovery
Procedure

Function: Accepts error message (either channel error or invalid BSTAT) and performs error recovery. Issues read sense data and read internal buffer requests. Makes a system log entry if necessary and wraps up error processing. This module contains the following procedures:

#LOTZERP	Error recovery procedure
LOTZMISC	Handle miscellaneous request I/O errors
REQIOERC	Common IOM detected error recovery
TZOPFAIL	Handle ORE with bad BSTAT and initiate error recovery
READSENS	Send read sense data request
TZERPROC	Handle read sense data and read internal buffers response
TZERCDCL	Decode DSTAT data
RDTZEBUF	Read internal error buffer
TZSYSLG	Build and send error log entry
TZCHNLER	Handle channel error message and initiate error recovery
LOTZCNLR	Process the channel error
LOTZCNLF	Finish processing the channel error
RTNCHMSG	Return channel error message
TERMERRP	Terminate error processing
LOTZCONT	Continue printing after error
LOTZRSOU	Restart the OU task
LOTZBFR	Build and send the feedback record
LOTZBFC	Build and return feedback record

How Invoked: Within the 3262/5211 printer IOM.

#LOTZIOM 3262/5211 Printer Mainline

Function: Provides the interface between the user procedure and the OU task performing the printer function. This is a queued interface where this module accepts requests on its input queue, performs the task (possibly performed by another module), and then responds to the request by returning the request to the appropriate queue. This module contains the following procedures:

#LOTZIOM	Initial activation of module
LOTZRIO	Handle request I/O
LOTZSCSP	Execute Standard Character Stream Print command
LOTZSEND	Send standard character stream print operation to OU queue
LOTZOUR	Handle OU task response
LOTZRSP	Handle standard character stream response
LOTZRSND	Resending the ORE to the OU task
LOTZSCSC	Standard Character Stream command completion processing
TZRSTART	Restart the OU task
LOTZSIDX	Handle segment identification crossings
LOTZINIT	Handle initialization
LOTZICLN	Internal cleanup routine for IMPI exceptions

How Invoked: Other VMC components.

#LOTZLUD 3262/5211 Printer Modify Logical Unit Description Request Handler

Function: Processes requests to modify LUD (logical unit description) by providing the interface between the user and the OU task performing the printer function. After performing the requested operation, a response is returned to the appropriate queue. This module contains the following procedures:

#LOTZLUD	Complete analyzing request
LOTZMLNM	Return a modify LUD message with a good status
LOTZMLMG	Return a modify LUD message
LOTZRMSG	Return message to return queue
LOTZQSC	Handle quiesce
LOTZRES	Handle resume
LOTZDAC	Handle de-activate
LOTZRCN	Handle request I/O continue
LOTZDSA	Handle change device-specific area
LOTZMLP	Handle modify LUD pending
LOTZDEVI	Build and send ORE to write device-specific area parameters to the adapter
LOTZMINL	Handle a forms length of 2 to 17 lines
TZLDSACK	Validate device-specific area parameters
LOTZRST	Handle reset
LOTZRSTF	Dequeue all request I/Os from IOM queue
LOTZVON	Handle vary-on
LOTZLUDR	Decode OU response to a modify LUD request
LOTZACT	Handle activate
LOTZVOFF	Handle vary-off
LOTZSSP	Handle suspend

How Invoked: Within the 3262/5211 printer IOM.

3203 Printer Local IOM

#LODRIOM 3203 Printer Mainline

Function: Provides the interface between the user procedure and the OU task performing the printer function. This is a queued interface in which this module accepts requests on its input queue, performs the task (possibly performed by another module), and then responds to the request by returning the request to the appropriate queue. This module contains the following procedures:

LODRIRIO	Validate the request I/O and build the print ORE
LODRISND	Send the print ORE to the OU task
LODRIOUR	Handle the OU task response
LODRISPR	Handle the standard character stream print response from the OU task and returns completion status to the user
LODRISID	Build an auxiliary FOB to handle data that crosses a segment boundary
LODRINIT	Initialize the 3203 IOM
LODRICLN	Handle exception handler cleanup

How Invoked: Other VMC components.

#LODRLUD 3203 Printer Modify Logical Unit Description Request Handler

Function: Processes requests to modify the LUD by providing the interface between the user and the OU task performing the printer function. After performing the requested operation, a response is returned to the appropriate queue. This module contains the following procedures:

LODRLGPM	Return the modify LUD message to the user
LODRLPML	Complete the pending modify LUD quiesce or suspend messages
LODRLQIS	Process the quiesce message
LODRLRSM	Process the resume message
LODRLDAC	Process the de-activate message
LODRLRN	Process the request I/O continue message
LODRLMDS	Verify the parameters in the device-specific area on a device-specific area change message
LODRLPND	Process pending device commands
LODRLDSA	Forward the device-specific area information to the printer
LODRLUPC	Build and send the Block/Allow Unprintable Character command to the device
LODRLOLD	Modify device to handle extended forms length
LODRLRST	Initiate processing of a reset message
LODRLRTR	Handle OU responses during reset processing
LODRLRTF	Finish reset processing and returns the reset message
LODRLU DR	Complete processing of the vary-on, activate, and vary-off messages
LODRLVON	Process the vary-on message
LODRLACT	Process the activate message
LODRLVFF	Process the vary-off message
LODRLSAD	Reset the SSADL flags and shut off the IOC
LODRLSUS	Process the suspend message

How Invoked: Within the 3203 printer IOM.

#LODRERP 3203 Printer Error Recovery Procedure

Function: Accepts error messages and performs error recovery. These errors include channel errors, unsuccessful completion of OU responses, post event errors, OU task exceptions, and invalid Request I/O instructions. This module also returns the error response to the user. This module contains the following procedures:

LODREREQ	Reject the Request I/O instruction because of an invalid field
LODRBFR	Build and send the feedback record
LODREOUE	Handle OU task responses when unsuccessful completion code occurs
LODRECBY	Process a response that encountered a channel busy condition
LODRESNS	Build and send error recovery OREs to the printer
LODREPRC	Handle responses to error recovery OREs
LODRERBF	Build and send OREs to read internal buffers
LODREPNR	Decode operator intervention required errors
LODREDCD	Determine error from the device status field
LODRELOG	Build and send error data to the system error log
LODREZLG	Clear the IOM internal error log data
LODRECHN	Process messages received from the channel IOM
LODREDCE	Decode the channel IOM message and determines if the message was caused by a post event error, FOB time-out, or a channel error
LODRECNR	Handle responses from the OU task after receiving a channel message
LODRETRM	Terminate error processing
LODRECON	Process Continue Print After Error command
LODRECOQ	Recall OREs from the OU task
LODRERSO	Restart the OU task

How Invoked: Within the 3203 printer IOM.



Load/Dump Management

INTRODUCTION

The load/dump management component is designed as a "pipeline", where the individual modules do a part of the total process, and the information passes from task to task in a direct line. Performance is greatly improved with this design concept because each module is a separate task. Work flowing through load/dump is not serialized while waiting for I/O, because other load/dump tasks can continue to run.

Load/dump management provides the functions that allow the user to save objects on an external medium such as, a diskette or an internal dump space, and retrieve those objects at a later time. The objects that can be processed by load/dump are:

- Data spaces
- Data space indexes
- Independent indexes
- Programs
- Journal spaces
- Space objects
- Dump spaces

If objects are dumped internally to a dump space, and the dump space is dumped to an external medium, the objects can be restored directly from the external medium.

The interface to Load/Dump is through the Request I/O, Modify LUD, and Request Path Operation instructions. REQIO and MODLUD are used when the operation is to external media (tape or diskette). REQPO is used when the operation is to an internal dump space. For simplicity in the following section, only the REQIO and MODLUD instructions are mentioned. The following chart shows the equivalent REQPO function:

REQIO and MODLUD

REQIO (normal)
REQIO (continue)
MODLUD Activate
MODLUD Reset
MODLUD Suspend
MODLUD Quiesce
MODLUD De-activate

Equivalent REQPO function

I/O Request
I/O Request (continue)
Initiate Path
Reset Path
Suspend Path
Quiesce
Terminate Path

These objects can be dumped, loaded over an existing object, or created and loaded onto this or another system. A load/dump session is initiated by a Modify Logical Unit Description (activate) instruction. The user communicates with load/dump through the Request I/O instruction.

The Modify Logical Unit Description instruction is used to change load/dump states (for example: suspend and reset). The Request I/O instruction is used to specify the load/dump commands and to specify the pointer to the objects to be processed.

Figure 26-1 shows an overview of load/dump management. Load/dump management consists of 11 modules, each with many internal subroutines. #LDPREP is invoked from #SIRQIO1, Request I/O instruction processes and checks for errors on the Request I/O instruction before it is sent to the load/dump pipeline modules. The remaining modules make up the load/dump pipeline. In the load/dump pipeline, each object that is to be loaded or dumped is broken into blocks of data to be processed by the diskette or tape IOM.

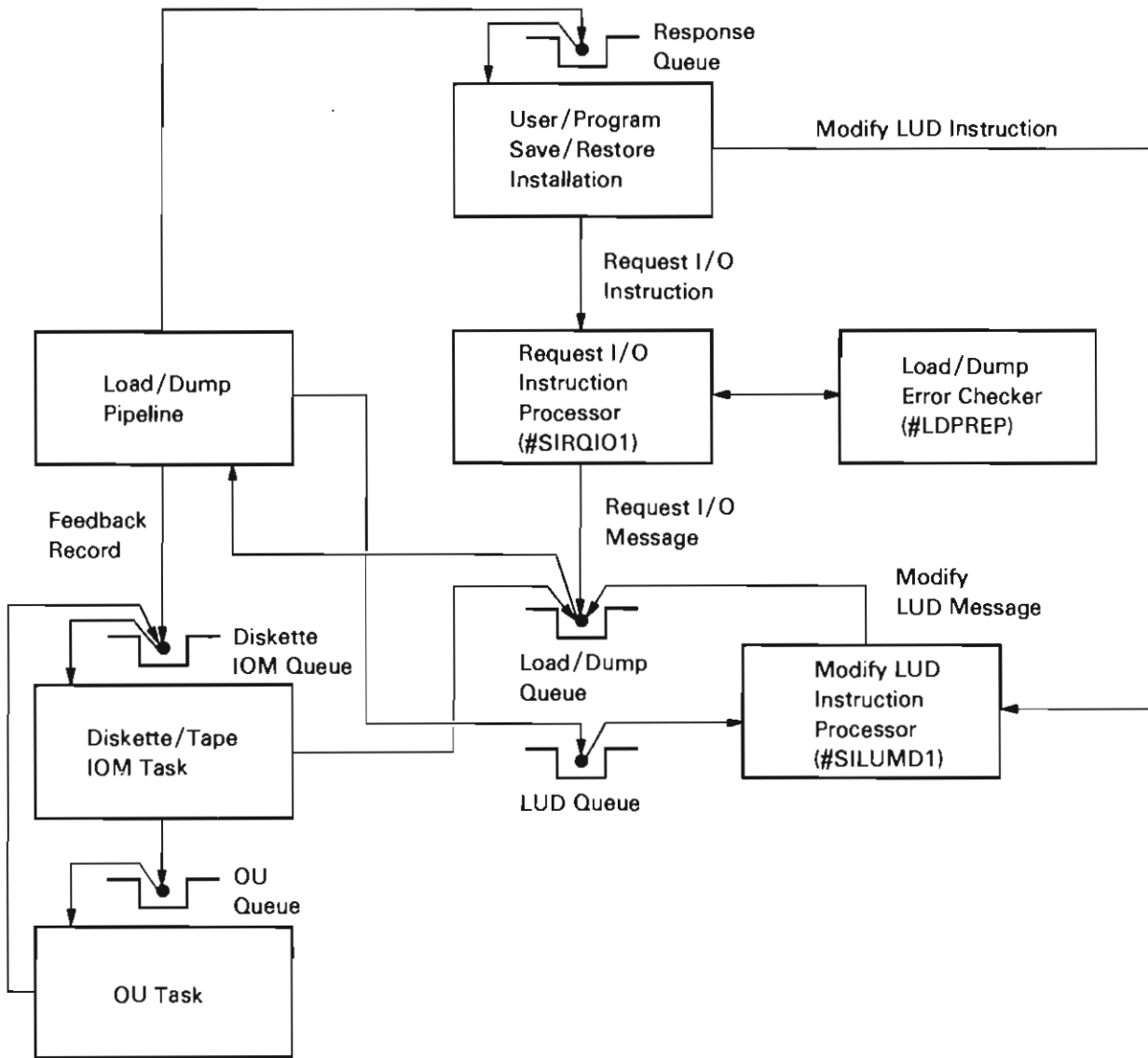


Figure 26-1. Load/Dump Processing

Load/dump processing is performed by a series of tasks. Each task in the "pipeline" performs a single major function of the total load/dump process. The major functions include, Request I/O instruction processing, load/dump network I/O preprocessing, object I/O preprocessing, object I/O processing, object I/O postprocessing, and load/dump network I/O postprocessing. The general flow of messages within the "pipeline", as shown in Figure 26-2, is from the Request I/O instruction processor through the I/O preprocessors, the object I/O processor, and the I/O postprocessors and then back to the Request I/O instruction processor.

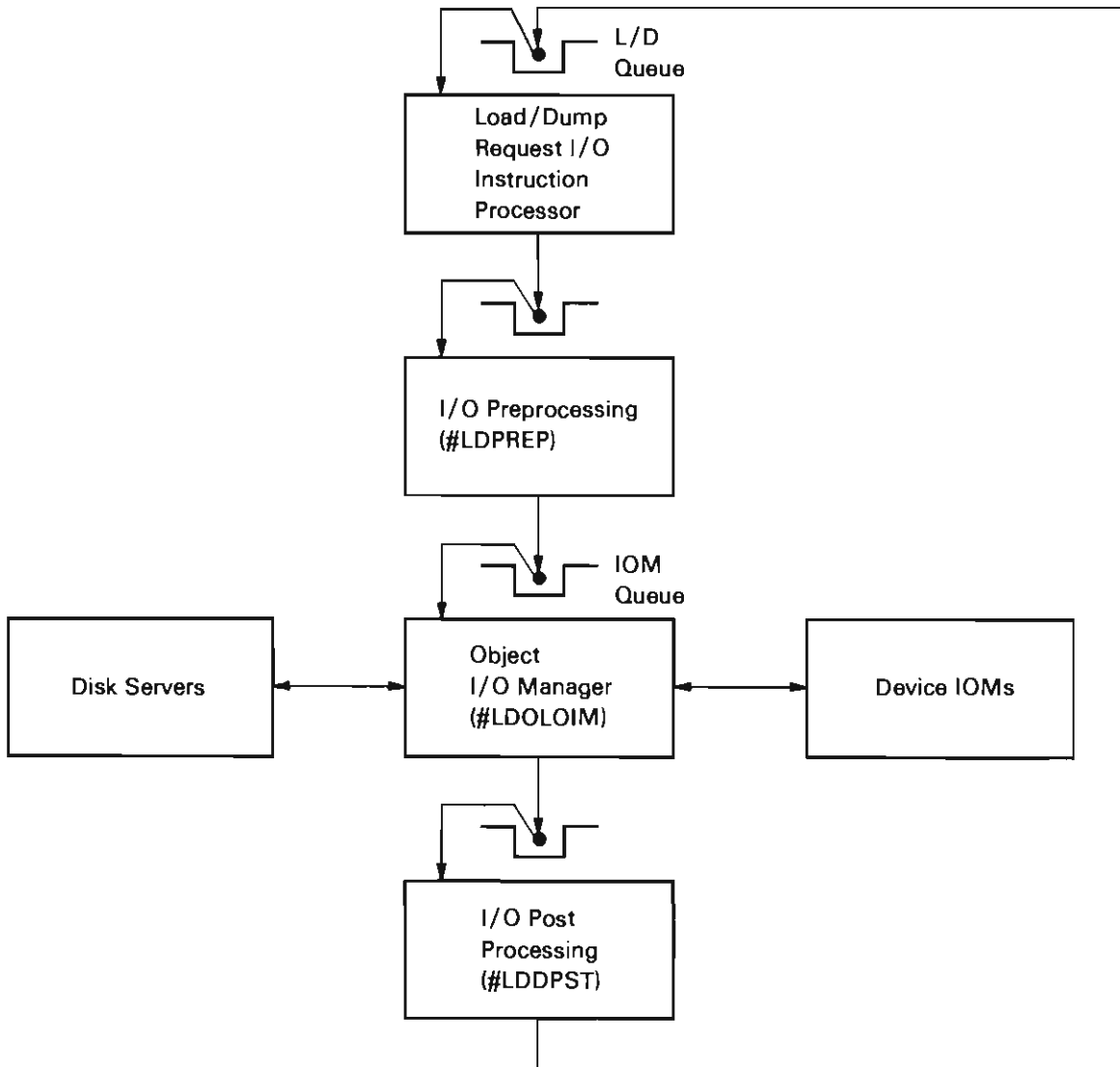


Figure 26-2. General Load/Dump Logic and Message Flow

Request I/O Instruction Processing

Error Checking the I/O Request

After load/dump is activated, the next logical step is for the user to issue an I/O request that contains load/dump commands. This I/O request is checked for errors that could occur at this time. The Request I/O instruction processor (#SIRQIO1) invokes #LDPREP to perform this check. #LDPREP uses various subroutines to validate the I/O request, the objects used, and the process and current invocation user profiles. If an error is detected, #LDPREP signals the appropriate exception. When all checks are performed and if no errors were detected, control is returned to #SIRQIO1. #SIRQIO1 sends a request I/O message to the load/dump queue for the load/dump "pipeline" to receive.

Dumping Objects

When #LDMODE receives an I/O request to dump objects, it scans through the request descriptor for network boundaries. Each time #LDMODE finds a network boundary, it formats a network message and sends the message to #LDDNPRE. The network message indicates which request descriptors are in the network, journal information, task dispatching element, and various counters and pointers for use by #LDDNPRE. Between each network message, the mode routine checks for modify LUD messages.

#LDDNPRE receives the network message from #LDMODE. #LDDNPRE seizes each object identified in the network message and transfers control to #LDDPST. For each dump request descriptor in the network, #LDDNPRE formats and sends an object message to #LDDOPRE. Each object message contains addressability to the specific request descriptor for that object, journal information, information on which network the object is from, and data areas for #LDDOPRE.

#LDDOPRE receives the object message from #LDDNPRE. An object-specific routine is invoked to verify that the object can be dumped. Another object-specific routine is invoked to build the load/dump descriptor, and to list the segments that are included in the object. #LDDOPRE adds a list of pointers to the object message indicating the segments to be dumped. This includes the load/dump descriptor segment and each of the segments in the object. The object message is then sent to #LDODOIM.

#LDODOIM receives the object message from #LDDNPRE. The segments listed in the message are brought into storage, pinned, and added to an address list element. The address list element is put into a message that is sent to the device IOM. The device IOM transfers the data to the media. When the process is completed, the segments are unpinned and the object message is sent to #LDDPST.

#LDDPST receives the object message from #LDODOIM, releases the object addressed by the message, and does any necessary journaling. The released object message is sent to #LDDNPRE to be reused. When #LDDPST has received all the object messages for a network, the network message is sent to #LDMODE. When all network messages have been sent and received, #LDMODE responds with a feedback record.

Loading Objects

To perform a load function, load/dump loads an object from a load/dump media either by overlaying an already existing object (load) or creating a new object (create and load). Load/dump searches the media for the request descriptor that describes the object to be loaded. When the correct request descriptor is found, load/dump either adjusts the object being overlaid or creates a new object in which to load the object. The object is then loaded. When the load function is completed, the object is restored to a usable form.

When #LDMODE receives an I/O request to load an object, #LDMODE scans the request descriptor for network boundaries. Each time a network boundary is found, #LDMODE formats a network message and sends it to #LDLNPRE. The network message contains information on which request descriptors are in the network, set command information, task dispatching element, and counters and pointers for use by #LDLNPRE. Between each network message, #LDMODE checks for modify LUD messages.

The load network preprocessing task (#LDLNPRE) is a function that receives a load network request from #LDMODE and breaks the network into load object requests to be processed by #LDOPRE (load object preprocessing task). The load network request has information about a group of request descriptors that are grouped together as a load/dump network.

When #LDLNPRE receives a load network request, #LDLNPRE scans the load/dump network to access the objects that are associated with the network. This involves doing an exclusive seize on each object associated with the load request and on each data space index that is over a data space associated with a load request. The seizes are transferred to #LDLNPST (load network postprocessing task), where each object is released after it is loaded.

#LDLNPRE then breaks the load/dump network into individual objects. #LDLNPRE formats a load object request for each load and create-and-load request descriptor in the network. Each load object request has information about the object to be loaded, as well as information about the network with which it is associated. As each load object request is formatted, it is sent down the "pipeline" to #LDLOPRE.

#LDLOPRE takes a load object request from #LDLNPRE and interfaces with storage management (#LDOLOIM) to load the object.

Work begins in #LDLOPRE when a load object request containing information about an object to be loaded from the media, is received. When an object on the media is to be loaded over an already existing object in storage, an object-specific routine is invoked to check the object in storage to make sure that it is a valid object to be loaded over. Once it is verified that the object can be loaded over, #LDLOPRE sends a read descriptor request to #LDOLOIM and waits for a response. After the request descriptor is read from the media (#LDLOPRE has received the read descriptor response), an object-specific routine is invoked to check the request descriptor to make sure that the object on the media is a valid object to be loaded into storage. When the request descriptor is verified, another object-specific routine is called to adjust the storage to the appropriate size necessary to contain the object to be loaded from the media.

#LDLOPRE then completes processing the load object request by building a read object request and sending it to #LDOLOIM. The read object request indicates that the object should be read from the media and loaded into storage.

#LDLOPST takes a read object response from #LDOLOIM and restores the loaded object to a usable form. Each object must have the encapsulated program architecture header and pointers to the associated space adjusted, as well as some object-specific processing, in order to put the object in a suitable format.

Each created and loaded object is inserted into a user profile. When requested, each object is also inserted into a context by #LDLOPST.

When it is requested that a create and load object should be journaled, #LDLOPST starts the journal process. An object-loaded entry is made for any object that is being journaled.

#LDLOPST then completes processing the object by sending a load object response to #LDLNPST, indicating that the object is in a usable form.

#LDLNPST receives load object responses from #LDLOPST and gathers them back into load/dump networks. When all the load object responses associated with a load/dump network are received, #LDLNPST completes network processing. Then links are restored between objects in the load/dump network. After the links are restored, the objects associated with the load/dump network are released. This task then completes its processing of the network by sending a load network response to #LDMODE indicating that the network is in a usable form.

When all network messages have been sent and received, #LDMODE responds with a feedback record.

Loading Objects from a Dump Space

Objects that are saved to an internal dump space, then dumped to an external medium, can be restored directly. The objects are restored from the external medium if the proper bit is set in the set load/dump parameters request descriptor.

When #LDPREP module verifies the source/sink request, it checks for a set load/dump parameters request descriptor with the 'load out of a dumped space' bit set. #LDPREP sets a corresponding bit in the load/dump session control block.

When #LDMODE receives an I/O request to load an object, it checks to see if the bit is set, and creates the #LDLOOC task. #LDLOOC (load out of container) checks to see if the media is positioned correctly. If not, an invalid descriptor error is signaled, and the I/O request ends. If the media is positioned correctly, #LDLOOC modifies the device parameters so #LDOLOIM builds its commands as if loading from a dump space. #LDLOOC also changes the address of the input queue for the IOM. Thus, #LDLOOC is now inserted between #LDOLOIM and the device IOM.

When #LDLOOC receives the read request, it sends a read request to the IOM to fill its internal buffers. #LDLOOC then copies data from its buffers to the buffer in the #LDOLOIM read request. Any data left over in #LDLOOC buffers is saved and copied to the next request from #LDOLOIM.

Modify Logical Unit Description Processing

Modify LUD Message Flow

The general strategy used by load/dump for handling Modify LUD instructions to the LUDs are as follows:

Note: There are some exceptions. These exceptions depend on the modify LUD messages and are listed under the appropriate modify LUD message.

- Load/dump receives the modify LUD message.
- Load/dump management processes the modify LUD message, informing the load/dump "pipeline" as needed. This may or may not involve broadcasting internal modify LUD type messages to the tasks in the "pipeline".
- When load/dump processing is completed, load/dump sends the modify LUD message to the device IOM. After processing is completed, the device IOM returns the modify LUD message to load/dump.
- When load/dump receives the response from the device IOM, the modify LUD message is returned to the modify LUD processor.

Activate Message

The device IOM creates the load/dump session when it receives a Modify LUD (activate) instruction for a primary LUD in load/dump mode. The device IOM creates the load/dump session control block and initializes the parts of the session control block that contain device-specific information. Then the device IOM creates the #LDMAIN task and passes the address of the session control block to a register. #LDMAIN provides the initialization and termination code for the load/dump session. The modify LUD activate message is then sent to #LDMAIN so that the activation status can be returned to MSCP along with the message.

#LDMAIN determines the load/dump mode (load or dump) and completes the initialization of the session control block for that mode. #LDMAIN then creates the queues and buffers required by load/dump for that function. When the queues and buffers are allocated successfully, #LDMAIN creates the "pipeline" tasks that are associated with that function. When activation processing is complete, #LDMAIN returns the modify LUD activate message and calls the mode routine (#LDMODE). The mode routine handles all active state processing.

Suspend Message

Load/dump suspends the load/dump session. This involves interrupting the Request I/O instruction in progress, but completing any networks in progress. The load/dump mode task stops sending work down the "pipeline" at the next network boundary. When all outstanding networks have completed, load/dump returns the feedback record with the suspended error status and the suspend message is returned with a successful status. If an error occurs, load/dump returns the feedback record with the appropriate error status and rejects the suspend operation.

Quiesce Message

Load/dump quiesces the load/dump session. This involves completing any Request I/O instructions in progress, and any Request I/O instructions on the input queue. The load/dump mode task keeps track of the functions that need to be completed. When all Request I/O instructions are completed, the quiesce message is returned with successful status. If an error occurs, load/dump returns the feedback record with the appropriate error status and rejects the quiesce operation.

Reset Message

When load/dump receives the modify LUD reset message from the modify LUD processor the following occurs:

1. Load/dump turns on a reset bit in the load/dump session control block. The reset bit is needed because parts of the "pipeline" may be slower in completing the reset, thus, a task may think it has been reset and then may receive some messages from an earlier task. The reset bit also aids in a quick start of reset. Tasks may start resetting immediately, even before they receive the reset message.
2. Load/dump broadcasts an internal reset message to all "pipeline" tasks. The internal reset message is sent to activate tasks that may be waiting on their input queues. The message is broadcast, instead of passed down the "pipeline", because a task waiting on an external seize may hold up the passing of the reset message.
3. Load/dump returns any outstanding retry messages to the task that received the error. If a "pipeline" task has a retry message outstanding, it waits for the message before doing any reset processing.
4. Load/dump then waits for outstanding work and reset responses to be returned. Any retry messages received now are returned to the proper task.

- Each task receives the internal reset message, then returns the message to the load/dump queue. If the task is in error mode, it waits until it receives a retry message and then aborts the current work.

When an abnormal end-of-task occurs, the task receives any input messages necessary to complete processing. The task then releases all seized and passes all current work through the "pipeline" with a partially processed completion code. No work is put on the input queue as long as the reset bit in the load/dump session control block is turned on. Any error message from a previous task is passed through unchanged as in normal processing.

- When all outstanding work is returned, and all the internal reset messages are returned, the mode routine returns the feedback record with a partially processed error summary set (only if there is an internal request I/O message). The mode routine then removes all the request I/O messages from the input queue and returns feedback record messages with the error summary. The reset bit in the load/dump session control block is turned off and the modify LUD reset message is returned to the modify LUD processor. This modify LUD reset message is not routed through the device IOM because the object IOM (the "pipeline" task that communicates with the device IOM) resets the device IOM as part of the object IOM reset processing.

Resume Message

Load/dump does no processing for a resume message. Only the state of the LUD changes.

Load/Dump Indicators – Exchange Message

Load/dump receives the modify LUD exchange message and changes the needed pointers so the noncurrent LUD and the current LUD are switched. The exchange message is not sent to the device IOM.

De-activate Message

The mode routine receives the message and sends it to #LDMAIN. #LDMAIN broadcasts an internal de-activate to each task in the "pipeline". Each "pipeline" task that receives the message cleans up, returns the message, and destroys itself. #LDMAIN deallocates all buffers allocated during session activation. #LDMAIN sends the modify LUD de-activate message to the device IOM and waits for it to be returned. When the response is received, #LDMAIN returns the de-activate message to the modify LUD message processor and destroys itself.

First Time-out Message Processing

If load/dump does not know of any modify LUD messages, either internally or at the device IOM, the time-out message is returned to the Modify LUD processor and no action is taken. If there is a modify LUD message at the device IOM, load/dump starts second time-out processing. See *Second Time-out Message Processing* later in this chapter.

If load/dump is processing a quiesce or suspend message, the message is rejected with time-out status. If load/dump is processing any other modify LUD messages, load/dump starts second time-out processing.

Second Time-out Message Processing

Note: Second time-out processing does not necessarily mean that a second time-out message was received. It is a name used for abnormal ending of the load/dump task.

Second time-out processing involves starting a timing process that causes load/dump to get a fake second time-out 5 seconds after a real second time-out message. The fake time-out message contains a nonzero return queue pointer. Load/dump issues the fake time-out because it never gets a real second time-out message if the device is a tape.

Load/dump receives the second time-out message (real or fake) and passes it directly to the device IOM. Any response is ignored by the load/dump.

Load/dump then begins to take down the session by broadcasting an internal message to time-out any subordinate tasks. Load/dump waits for all the internal time-out messages to be returned, and then #LDMAIN destroys itself.

Error Handling

When any "pipeline" task receives an error, it stops processing new requests from its input queue by adjusting the receive key. The "pipeline" task builds an error message (using the current request message) that contains a completion code showing what error occurred, and information necessary to restart the task. This information always includes a pointer to the "pipeline" task input queue, and a copy of the current return queue address. This error message is then sent through the "pipeline".

Each task receives the error message after previous messages on its input queue are processed. The error message is not processed, but is sent through the "pipeline". The mode routine receives the error message and reads the completion code. The mode routine sends back a feedback record with the appropriate error summary.

If the I/O request is retried, the mode routine changes the error message into a retry message by restoring the return queue pointer from the saved copy, changing the key, and clearing the completion code field. Then the mode routine sends the message to the receiving queue of the task that received the error. When the failing task receives the retry message, it is released from error mode. The task then uses the information in the message to retry the request.

If an error occurs in two "pipeline" tasks at the same time, the error from the task working on the earliest request is processed first. Since that task has closed its input queue, it does not receive the error message from the other task until after the mode routine receives the error message, recovers and retries by sending the retry message to the task. The second error message continues through the "pipeline".

Storage Management for Load/Dump

Load/dump manages its own I/O and storage pool separately from the rest of VMC.

When the load/dump session starts, #LDMAIN sets the size of a special pool that is created out of the user's storage pool. This special pool is then used only for storage needed by load/dump. The size of the special pool is based on the size of the original user pool and the transfer size of the device. The special pool is created by the object IOM when the load/dump activate message is received. The object IOM returns the special pool pages to the user pool when a modify LUD de-activate message is received.

Data is manipulated only in the special pool pages. The use of the pages is explicitly managed by the object IOM. Access to secondary, or disk storage is also managed explicitly by the object IOM. The normal storage management interfaces are not used. When the object IOM is activated, a variable number of disk server tasks are created. The number is set by #LDMAIN and depends on the number of disks on the machine. These disk server tasks handle all the accesses to the disks for the load/dump session for data being loaded or dumped. Normal paging continues to use the normal storage management. When the object IOM receives the de-activate message, the disk server tasks are destroyed.

Device IOM

Load/Dump Session Control Block (Create/Destroy)

A load/dump session control block is created by a diskette or tape IOM each time a device LUD is varied on and is destroyed when the device LUD is varied off. This is to ensure that a load/dump queue, contained in the load/dump session control block, exists when an activate message is received for a load/dump session and also to ensure that the load/dump session control block and load/dump queue exist until a load/dump session is de-activated. The load/dump session control block must exist before activation and after de-activation of a load/dump session so the modify LUD activate, de-activate, and time-out messages are handled properly.

Load/Dump Session (Activate/De-activate)

A common routine (#LOLDDAD) is called by the device IOMs to activate or de-activate a load/dump session. To activate a primary load/dump session, #LOLDDAD initializes the load/dump session control block including the load/dump queue contained in the load/dump session control block. #LOLDDAD then creates the load/dump task. If the create task is successful, the activate message is forwarded to the load/dump queue and returned to the Modify LUD processor by load/dump management. If the create task fails, the activate message is returned to the Modify LUD processor with the error status. To activate an alternative load/dump session, the routine copies the load/dump queue pointer and load/dump session control block pointer from the primary LUD to the alternative LUD and returns the activate message to the Modify LUD processor.

After the primary load/dump session is activated, all modify LUD messages for the current load/dump session are sent to the load/dump queue. All modify LUD messages for noncurrent load/dump session devices are sent to the IOM queue.

If a time-out message is received by the device IOM for the activation of the primary load/dump session, it is forwarded to the load/dump queue.

The common routine (#LOLDDAD) is again invoked by the device IOM to de-activate a load/dump session device. The pointers to the load/dump queue and the load/dump session control block are replaced by zeros, and #LOLDDAD then returns the de-activate message to the sender.

DATA AREAS

Session Control Block

There is one session control block for each load/dump session. The session control block is the common structure for intratask control and communication. The session control block is divided into four areas: general, load/dump specific, object I/O manager specific, and device I/O manager specific.

The general area of the session control block contains pointers and session tuning information. Some of the pointers and tuning parameters are created by the device IOM during session activation. The rest are created by #LDMAIN, during session activation, and updated by #LDMODE during modify LUD processing.

The pointers point to structures and queues outside the load/dump "pipeline". The device IOM input queue pointer and current LUD pointer are set by the device IOM during session activation. The load/dump session input queue pointer is set by #LDMAIN during session activation. The current LUD pointer and current modify LUD message pointer are updated by the #LDMODE during modify LUD processing.

The tuning parameters control the performance of the load/dump "pipeline". There are three types of tuning parameters: LUD parameters, device parameters, and load/dump parameters. The LUD parameters are specified by the user when the Modify LUD (activate) instruction is issued. The device IOM copies the parameters from the LUD to the session control block during session activation. The device parameters are values defining the data transfer characteristics of a specific device. The load/dump parameters are values calculated from the LUD and device parameters. The load/dump parameters define the limits on the resources used by load/dump during an active session.

The load/dump specific area is initialized by #LDMAIN during session activation. The load/dump specific area of the session control block contains queues and pointers that control the flow of information within the load/dump "pipeline". There is one queue for #LDMAIN and #LDMODE, and one for each "pipeline" task in a session. The control information is contained in two tables: the "pipeline" communications table and the flow control table. The "pipeline" communications table supports the intertask communication that bypasses the normal sequential flow of the "pipeline". The flow control table contains one flow control entry for each task in the "pipeline". The flow control table controls the flow of information in the "pipeline" by specifying for each task the queue that it uses for input, and the queue it uses for output.

Dump Network Message

The dump network message is created by #LDMAIN. #LDMODE formats the dump network message and sends it to #LDDNPRE. The network message indicates where the network starts, how many request descriptors there are in the network, how many objects are in the network, journal information, and task dispatching element.

Each object message has a pointer to its dump network message. #LDDPST uses this pointer to locate the dump network message. As each object message is received, #LDDPST increases an objects-received count in the dump network message. When the objects-received count is the same as the number of objects in the network, #LDDPST sends the dump network message to #LDMODE.

Dump Object Message

The dump object message is created by #LDMAIN and formatted by #LDDNPRE. The dump object message contains journal information, an area for a segment identifier table, and pointers to the request descriptor, the object, the dump network message for the object, and the load/dump descriptor segment identifier. #LDDNPRE creates one load/dump descriptor segment identifier for each dump object message. The area for the segment identifier table is at the end of the dump object message. It contains a list of segments that are in the object. This table is built by #LDDOPRE, and identifies to #LDODOIM what segments to send to the device.

Load Network Messages

The load network message is created at initialization time by #LDMAIN. The segment identifier that contains this message also contains two tables associated with the network. These tables are the object table and the seize table. #LDMODE builds the load network message and sends it to #LDLNPRE. Based on information in the load network message, #LDLNPRE builds the two tables and places them at the end of the network message.

There is one entry in the object table for each object in the network. All the information to build the object table entry for a load operation is contained in #LDLNPRE. #LDLNPRE normally builds the object table. However, for a create and load operation, most of the necessary information is contained in the load/dump descriptor, which load/dump cannot access until the information is read from the media. In this case, #LDLOPRE builds the object table entry. Each entry in the object table has an indicator that is set on by #LDLOPST when an object is successfully loaded.

There is one entry in the seize table for each object in the network and one entry for each data space index that is over a data space being loaded in the current network. The seize table entries are built when each object is being seized by a load/dump "pipeline" task. #LDLNPRE seizes all objects being loaded over and builds the seize table entry at that time. For create and load operations, the object is seized by #LDLOPST after the object is loaded from the media, and the seize table is built at that time. Each entry has an indicator, indicating if the object is seized or released.

Addressability to the network message is established through each object message associated with the network, and #LDLNPRE gains addressability to the network message through the object messages. When #LDLNPRE has completed all processing on the network, it sends the load network response back to #LDMODE after releasing all the seized objects in the network.

Load Object Message

The load object message is contained in a segment identifier created during initialization by #LDMAIN. An object information segment associated with each load object message is created by #LDLNPRE, and a pointer to the object information segment is set in the load object message. The load object message invokes different processing in the "pipeline", depending on the value of the function code. The same message is used as a load object, a read descriptor, and a read object message by different tasks in the "pipeline".

The object information segment identifier contains the object segment table, the fix-up data, and the base page buffer that is associated with the object. The object segment table contains an entry for each segment in the object. The fix-up data varies from object to object. Some object types have no fix-up data, while with other object types, the length of the fix-up data depends on the characteristics of each particular object. Each object has a base page buffer that must begin on a page boundary and have the length of one page. Because the length of the object segment table and the fix-up data varies from object to object, the pointers to the data areas in this segment identifier are kept in the object message and are computed for each object.

There is one entry in the object segment table for each segment in the object. The object segment table is built by the object-specific routines that perform object storage adjustment. The object segment table is used to tell #LDOLOIM where the object is to be loaded and when the object is read from the media. The object segment table is found in the object information segment that is associated with each object message.

Recoverable Error Processing

When a recoverable error such as an end-of-volume error occurs, load/dump raises the receive key such that only request I/O continue or modify LUD messages can be received. Load/dump then returns to the user the request I/O that incurred the error. The user must then correct the error (for an end-of-volume this means that the user must interface directly with the IOM to reposition the device at the start of the next volume). The user then issues the Request I/O instruction that incurred the error to load/dump. This request must be put at the top of the load/dump queue. The user then issues a Request I/O (continue) instruction. Load/dump receives the Request I/O (continue) instruction, lowers the receive key to allow normal processing, and then returns the request I/O continue status to the user.

The next request I/O received by load/dump will be the request that incurred the error. Load/dump checks the RDs received with the request to detect that no modifications were made. Then load/dump passes the message to the IOM that is responsible for completing the request.

STRUCTURE

The following is a list of the modules in load/dump management and the function that each module performs. This list also shows how the module is invoked.

#LDCRTDS Create Dump Space

Function: Creates a new dump space according to the input specifications.

How Invoked: Within this component.

#LDDESDS Destroy Dump Space

Function: Destroys the specified dump space.

How Invoked: Destroy Dump Space instruction.

#LDDFDT Dump Flow Definition Table

Function: Describes the tasks and the order of those tasks in the "pipeline" for dump mode.

How Invoked: Within this component.

#LDDNPRE Load/Dump Dump Network Preprocessing Routine

Function: Preprocesses dump requests on a load/dump network. Distributes a network request to object requests.

How Invoked: Within this component.

#LDDOPRE Load/Dump Object Preprocessing

Function: Interfaces with the object IOM (#LDODOIM) to dump objects to a media.

How Invoked: Within this component.

#LDDPST Load/Dump Dump Object/Network Postprocessing Routine

Function: Sends dump object responses until a network is completed, then sends the dump network response.

How Invoked: Within this component.

#LDENSUR Ensure Dump Space Object Instruction Processor

Function: Ensures that all the data in the specified dump space is written to auxiliary storage.

How Invoked: Within this component.

#LDINSDD Insert Dump Data Instruction Processor

Function: Adds the given data record to the specified dump space.

How Invoked: Insert Dump Data instruction.

#LDLD Dump Space Object Specific Routine

Function: Verifies and provides information about a dump space when the dump space is being dumped or loaded.

How Invoked: Within this component.

#LDLDESC Validate Load/Dump Descriptor for Read Descriptor

Function: Verifies that the block of data read from the media is a load/dump descriptor and checks for a match with the requested object identifier. Returns additional information to the caller such as descriptor size, data format, block size when dumped, and space block count to the next descriptor.

How Invoked: Within this component.

#LDLFDL Load Flow Definition Table

Function: Describes the tasks and the order of those tasks in the "pipeline" for load mode.

How Invoked: Within this component.

#LDLFND Validate Load/Dump Descriptor for Find Next Descriptor

Function: Verifies that the block of data read from the media is a load/dump descriptor and copies the object identifier into the specified object identifier buffer. Returns additional information to the caller such as descriptor size, data format, block size when dumped, and space block count to the next descriptor.

How Invoked: Within this component.

#LDLNPRE Load/Dump Load Network Preprocessing Routine

Function: Preprocesses load requests on a load/dump network. Sends a network request into object requests.

How Invoked: Within this component.

#LDLNPOST Load/Dump Load Network Postprocessing Routine

Function: Does load postprocessing on a load/dump network. Distributes object responses until a network is completed, then restores the network to a usable form.

How Invoked: Within this component.

#LDLOOC Load Objects from a Dump Space

Function: Reads the data from the media and presents it to load/dump management as needed. This routine maps the block sizes that are on the media.

How Invoked: Within this component.

#LDLOPRE Load/Dump Load Object Preprocessing Routine

Function: Interfaces with the object IOM (#LDOLOIM) to load objects from a media.

How Invoked: Within this component.

#LDLOPST Load/Dump Load Object Postprocessing

Function: Restores an object to a usable form after it is loaded from a media.

How Invoked: Within this component.

#LDMAIN Load/Dump Session Main Task Entry Point

Function: Provides the common initialization and termination code for the load/dump session.

How Invoked: Within this component.

#LDMATDS Materialize Dump Space Attributes Instruction Processor

Function: Retrieves and materializes the attributes of the specified dump space.

How Invoked: Within this component.

#LDMODDS Modify Dump Space Attributes Instruction Processor

Function: Modifies the attributes of the specified dump space.

How Invoked: Within this component.

#LDMODE Load/Dump Mode Routine

Function: Controls the interface processing for load/dump in dump and load mode.

How Invoked: Within this component.

#LDODOIM Load/Dump Object I/O Manager for Dump

Function: Manages transfer of data from the single level storage to an external device for load/dump.

How Invoked: Within this component.

#LDOLOIM Load/Dump Object I/O Manager for Load

Function: Manages transfer of data from the external device to single level storage for load/dump.

How Invoked: Within this component.

#LDPIPEH Load/Dump "Pipeline" Default Exception Handler

Function: Handles and reports unexpected exceptions in load/dump "pipeline" tasks.

How Invoked: Within this component.

#LDPREP Load/Dump Request I/O Preprocessing Routine

Function: Preprocesses error checking for the load/dump I/O request. The error checking is done synchronously by the process.

How Invoked: Request I/O instruction.

#LDREQPO Request Path Operation Instruction Processor

Function: Provides the interface between the REQPO instruction and load/dump.

How Invoked: Within this component.

#LDRETDD Retrieve Dump Data Instruction Processor

Function: Gets a data record out of the specified dump space at the specified location.

How Invoked: Within this component.

#LDSIDS Change Dump Space Size Routine

Function: Extends or truncates the dump space data area as requested.

How Invoked: Within this component.

#LODDCD Load/Dump Data Compression/Decompression Task for 72MD Diskette Device

Function: Compresses the data before sending it to the 72MD IOM task for dump, and decompresses the data from the 72MD when restoring it.

How Invoked: Within this component.



Machine Services Control Point

INTRODUCTION

The machine services control point (MSCP) consists of routines that provide services to other source/sink components. These routines assist in allocating and controlling the use of all source/sink resources. The functions of these routines are to:

- Establish the physical and logical paths over which a user can communicate with a source/sink device.
- Assist in recovery and orderly termination of a session when a failure occurs.
- Handle requests on the MSCP-logical unit (LU) and MSCP-physical unit (PU) sessions, SDLC only.

There are two types of MSCP routines.

- **Synchronous routines:** These routines execute synchronously to the requester, are invoked using a call/return interface, and execute under the process of the requester.
- **Asynchronous routines:** These routines execute under an independent MSCP task and perform their functions asynchronous to any processing done by the requester.

Both types of routines use the source/sink active device list. The source/sink active device list is a control block that contains the information required by these routines. This block is further described in the *Data Areas* in this section.

The System/38 instructions and functions supported by the synchronous MSCP routines are as follows:

- **Modify Controller Description**
 - Vary-on
 - Dial
 - Abandon connection
 - Vary-off
 - Continue
 - Cancel
- **Modify Logical Unit Description**
 - Vary-on
 - Activate/de-activate session
 - Vary-off
 - Continue
 - Cancel
- **Modify Network Description**
 - Vary-on
 - Enable
 - Manual answer/manual start data
 - Disable
 - Abandon call
 - Vary-off
 - Continue
 - Cancel

The synchronous MSCP routines are invoked by the user process, and they complete execution before the System/38 instruction is complete.

The MSCP task (asynchronous) is a VMC function that is always active, waiting on a queue for function requests or responses to MSCP messages. When a message is received, the base MSCP module (#MSCPTSK) routes the message to the appropriate routine according to the function code. These routines do not wait for any asynchronous requests or responses; they perform their functions and return control to #MSCPTSK.

Modify Controller Description (Synchronous)

The MSCP routines that perform the modify controller description (CD) functions for vary-on communications and dial do not perform the entire function. In these cases, messages are sent to the appropriate I/O manager (IOM) to initiate the function. At some time later, additional asynchronous processing by the MSCP task is required to complete a vary-on and dial.

Vary-On

#MSCVONN acquires storage and builds the CD source/sink active device list control block. For nonswitched lines, the CD is associated with a specific network description (ND) that must already be varied on. The CD source/sink active device list block is created, and a message is enqueued to the CD pending queue in the source/sink active device list. A request exchange identification message is then sent to the SDLC/BSC/MTAM IOM to request that the station be polled for the exchange identification information for that station.

Note: For a 3274 Controller, a request contact message is also sent to the asynchronous MSCP task to start the contact sequence.

This ends the synchronous processing and the CD is set to the vary-on pending state.

When a CD for a nonswitched network only is being explicitly varied-on using the Modify CD instruction and the ND associated with the CD is not available due to prior failures or the reuse of the communications line has been suspended, the recovery resource/activation state for the CD is set to the status of the ND. The CD is in the varied-on pending state and synchronous processing is completed.

For switched lines, the CD can be associated with any number of NDs when a call is completed. The CD source/sink active device list block is created, and a message is enqueued to the CD pending queue in the source/sink active device list until the connection is made. The CD is set to the varied-on pending state, ending the synchronous portion on the vary-on.

For directly attached CDs (no NDs), the complete vary-on process is performed. This consists of the following:

- The operational unit (OU) tasks and the IOM task are created and their registers initialized. This is not done for a display station pass-through controller.
- The input queues are allocated to these tasks.
- A CD contact message is sent to the IOM. This is not done for a display station pass-through controller.
- A CD source/sink active device list block is created and chained to the source/sink active device list.
- A CD contact successful event is signaled for the work station controller or a display station pass-through controller.

For X.25 permanent virtual circuits (PVC), the CD is associated with a specific ND and a logical channel entry. When the CD is varied on, the MSCP creates and initializes the CD source/sink active device list control block. The CD pending message is enqueued to the CD source sink active device list control block. The key of this message contains the remote XID, the local line ID, the logical channel ID, and an indication that this pending message is for an X.25 CD. A request XID message is sent to the XIOM to send XID across the network. The request XID message contains the CD address for a PVC. When the request XID message is returned, the synchronous processing of vary-on CD is complete and the CD status is set to the vary-on pending state.

For X.25 switched virtual circuits, the CD could be associated with any of a number of NDs when the call is completed. Also, the CD could be associated with any number of logical channel entries when the call is completed. When a switched CD is varied on, a CD source sink active device list control block is created and initialized. Also, the CD pending message is enqueued to the CD pending queue. The key of this message contains the remote network address, the remote password, and an indication that this is an X.25 CD. The CD is set to vary-on pending state and the synchronous processing is complete.

Dial

#MSCDIAL processes the modify CD dial request. The ND candidate list in the CD is searched to locate a switched ND that is in the switched-enabled state for dial out with the recovery resource/activation state set to continue. The line is then allocated to the dial operation for this CD. An abandon call message is sent to the MSCP task to inhibit the SDLC/BSC/MTAM IOM from accepting incoming calls. If the status field in the abandon call message is good, an initialize-line message is sent to the SDLC/BSC/MTAM IOM. Dialing is in progress when good status is returned with the initialize-line message. If the ND for the line indicates that the machine is in autodial mode, then the connect-out message is sent to the SDLC/BSC/MTAM IOM. For manual dial, the CD manual intervention event is signaled with system pointers to the CD and the ND. The CD and ND are set to the dialing state, ending the synchronous portion of the dial.

For X.25, #MSCDIAL processes the modify CD dial request. The ND candidate list in the CD is searched to locate an X.25 ND that has at least one switched virtual circuit out or switched virtual circuit both logical channel entry available. If a logical channel is available, the MSCP sends the connect out message to the XIOM to initiate an outgoing call request. The connect out message is sent to the XIOM with the address of the CD in it. There may not be an outgoing logical channel available when the outgoing call request gets to the adapter. If there is no outgoing logical channel available, the connect out message is returned to the MSCP with an appropriate status code. The MSCP then signals the CD contact unsuccessful event with a reason code indicating that there was a dial failure.

Abandon Connection

#MSCABAN (an entry point in #MSCVONN) processes the modify CD abandon connection request. This module sends the abandon call message to the MSCP task to terminate the connection.

A message is enqueued to the CD-pending queue in the source/sink active device list. The CD is then set to the varied-on-pending state.

For X.25, the modify CD abandon connection instruction is used to terminate a switched virtual circuit. The reset LUD and vary-off LUD are sent to the station IOM for any LUDs varied on, then the reset CD message is sent to the station IOM. After an appropriate response is received to the reset CD message, vary-off is sent to the station IOM and discontact is sent to the XIOM. The abandon connection message is not sent to the XIOM. The CD pending message is again enqueued to the CD pending queue and the CD and LUDs are set to the varied-on-pending state.

Vary-Off

#MSCVOFF (an entry point in #MSCVONN) processes the modify-CD-vary-off request. The procedure of a vary-off is similar to that for an abandon connection with the following exceptions:

- Logical unit descriptions (LUDs) are always varied off before a CD is varied off.
- The CD is set to the vary-off state.
- The source/sink active device list CD block is unchained from the ND block or the CD pending list, and the associated storage is freed.

For X.25, the modify CD vary-off instruction is used to terminate a permanent virtual circuit or a switched virtual circuit. The reset CD message is sent to the station IOM. After an appropriate response is received to the reset CD message, vary-off CD is sent to the station IOM and the discontact message is sent to the XIOM. The abandon connection message is not sent to the XIOM for a switched or permanent virtual circuit. The CD source/sink active device list control block storage is freed and the CD is set to varied-off state.

Cancel

#MSCCNCL (an entry point in #MSCVONN) processes the modify CD cancel request. This routine is used to suspend the reuse of the controller and any attached varied-on LUDs by setting the recovery resource/activation state in the CD and LUD objects to cancel. If the CD has varied-on LUDs attached, the recovery resource/activation state is set to cancel in each attached LUD object.

For all attached LUD objects, all queued pending activate resource request I/Os present when this instruction is issued are returned to the machine interface response queue.

Continue

#MSCCNTU (an entry point in #MSCVONN) processes the modify CD continue request. This routine is used to enable the reuse of the controller and any attached varied-on LUDs by setting the recovery resource/activation state in the CD and LUD objects to continue. If the CD has varied on LUDs attached, the module #MSLCNTU is called for each LUD to complete the continue processing.

For nonswitched lines or X.25 permanent virtual circuits, the CD is associated with a specific ND that must have a recovery resource/activation state of continue or active. A message is enqueued to the CD pending queue in the source/sink active device list and the request exchange identification message is sent to the SDLC/BSC/MTAM IOM requesting that this station be polled to get the station's exchange identification information.

Note: For a 3274 Controller, a request contact message is also sent to the asynchronous MSCP task to start the contact sequence.

For switched lines or X.25 switched virtual circuits, a message is enqueued to the CD pending queue in the source/sink active device list and when an activate resource request I/O is present for any attached LUD, the CD switched intervention event is signaled with a system pointer to the CD. The synchronous continue processing is complete with the remainder of the function to be performed when an incoming call is received or a Modify CD (dial) instruction is issued.

For directly attached work station controllers, all of the MSCP functions needed to complete the continue are performed synchronously. This consists of the following:

- A CD contact (ERP) message is sent to the native IOM.
- A CD contact successful event is signaled for the work station controller and a display station pass-through controller.
- A vary-on LUD continue message is sent to the MSCP task for each varied-on LUD attached to the CD.

Modify Logical Unit Description (Synchronous)

Vary-On

#MSLVONN processes the modify-LUD-vary-on request. The function performed by #MSLVONN is determined by the type of LUD being varied on.

For a local device, the entire vary-on processing is completed by #MSLVONN. This is accomplished as follows:

- The OU task and IOM task are created, and the registers are initialized.
- The input queues are allocated to these tasks.
- A vary-on LUD message is sent to the IOM.
- An LUD source/sink active device list block is created.
- When the IOM responds to the varied-on message, the MSCP function is completed by setting the LUD to the varied-on state and signaling a LUD-contact successful event.

For a device attached through a CD, the LUD state can only be advanced to the state of the CD. For example, if the CD is in the varied-on pending state, the LUD can be advanced only to the varied-on pending state. When the CD is advanced to the varied-on state, the LUD can be advanced to the varied-on state and the process completed. For a LUD attached to a display station pass-through CD, the LUD goes to a varied on state (recovery resource/activation state indicates normal pending).

When the LUD is attached to a CD, there are no tasks to be created because the IOM task is associated with the CD for SDLC/work station or the ND for BSC or MTAM. An LUD source/sink active device list block is created and a vary-on-LUD message is sent to the MSCP task. Satisfactory response to this message completes the MSCP processing for a vary-on-LUD operation.

When a LUD attached to a CD is being varied-on and the CD is not available due to prior failures, or the reuse of the controller has been suspended, the recovery resource/activation state for the LUD is set to the status of the CD.

Activate/De-activate

When a modify LUD to activate or de-activate a session is executed, #MSLSCRT and #MSLSDES send the appropriate message to the proper IOM.

For a LUD attached to a BSC CD that is being activated and the MSCP recovery for the device has not completed, the activate session is rejected with the error status that caused the inoperative condition. The LUD remains in the varied-on state and the synchronous processing is complete.

Vary-Off

#MSLVOFF (an entry point in #MSLVONN) processes the modify LUD vary-off request. A vary-off operation is similar to a vary-on operation. The type of LUD and the state of the LUD determine the processing performed. Each vary-on step has a corresponding vary-off step. The following steps are performed as required to vary-off an LUD:

- A vary-off LUD message is sent to the MSCP task to reset the MSCP-LU data flow if the LUD is attached to a CD. For a locally attached device, the vary-off LUD message is sent to the IOM.
- A De-Activate Logical Unit Systems Network Architecture (SNA) command is sent to the device if required.
- A vary-off LUD request message is sent to the MSCP task if the LUD is attached via a CD. For a LUD attached to a display station pass-through CD, the vary-off LUD message is sent to the IOM only if the pass-through task is active. For a local device, the vary-off LUD message is sent to the IOM.
- The tasks are destroyed.
- The queues are returned to the queue pool.
- The source/sink active device list LUD block is freed.

Cancel

#MSLCNCL (an entry point in #MSLVONN) processes the modify LUD cancel request. This routine is used to suspend the reuse of the device by setting the recovery resource/activation state in the LUD to cancel.

All queued pending activate resource Request I/O instructions present when this instruction is issued are returned to the response queue.

Continue

#MSLCNTU (an entry point in #MSLVONN) processes the modify LUD continue request. This routine is used to enable the reuse of the device by setting the recovery resource/activation state in the LUD to continue. When activate resource Request I/O instructions are pending, the recovery resource/activation state is changed to activation pending. The vary-on LUD continue message is sent to the MSCP task.

Modify Network Description (Synchronous)

Vary-On

When a modify ND vary-on request is executed, the MSCP creates the following tasks and initializes the associated registers:

- A transmit OU task
- A receive OU task
- An SDLC/BSC/MTAM/XIOM IOM task

The input queues for these tasks are allocated and the tasks are dispatched. If the first line is being activated on an IOC, the MSCP creates a wakeup task and sends a message to that task. The task invokes the wakeup routine (#TPLIOCO). This routine resets the I/O controller (IOC), performs some diagnostic functions, and issues the commands to put the IOC in normal mode.

For X.25, #TXIOPUP is called to send the appropriate startup commands to the IOP. If this is the second line to become active for this IOP, the adjust buffer allocation message is sent to the XIOM running for the first active line.

An activate-link message is then sent to the SDLC/BSC/MTAM/XIOM IOM.

If the random access memory load flag in the machine configuration record is set on, the MSCP loads the communications random access memory.

The MSCP provides protection for a given subsystem to ensure that a high speed line is not varied on concurrently with any other line or lines that share the same communications IOC.

Enable

#MSNENAB (an entry point in #MSNVONN) processes the modify ND enable request. If the ND being varied on represents a switched line with autoanswer, then initialize-line and connect-in messages are sent to the SDLC/BSC/MTAM IOM. The connect-in message response is sent to the MSCP task input queue when the autoanswer function is complete.

This instruction is not supported for X.25.

Manual Answer/Manual Start Data

These functions are requested as a result of a Modify ND instruction and are used to synchronize VMC controlled hardware adapter signals with the actions being performed by the operator in completing a switched connection.

Manual answer indicates that the operator has answered the phone and has established the connection.

#MSNVONA (an entry point in #MSNVONN) sends an initialize-line message to the SDLC/BSC/MTAM IOM to begin line adapter initialization.

Manual-start-data indicates that the operator has placed the coupler in data mode and the line is ready to be used for data communications. #MSNVOND (an entry point in #MSNVONN) sends a connect-in message to the line IOM. The SDLC/BSC/MTAM IOM signals the adapter to enable switched connection to begin line usage.

These instructions are not supported for X.25.

Disable

#MSNDSAB (an entry point in #MSNVONN) sends the abandon call message to the MSCP task to inhibit the SDLC/BSC/MTAM IOM from accepting incoming calls.

| This instruction is not supported for X.25.

Abandon Call

#MSNABAN (an entry point in #MSNVONN) is used to terminate a dial or answer connection. #MSNABAN sends the abandon call message to the MSCP task to terminate the connection.

| This instruction is not supported for X.25.

Vary-Off

#MSNVOFF (an entry point in #MSNVONN) processes the modify ND vary-off request. The Modify ND instruction to vary-off a network description initiates a request to the MSCP to free all resources allocated to the associated line. The MSCP sends a de-activate-link message to the SDLC/BSC/MTAM IOM. This causes the SDLC/BSC/MTAM IOM to reset the hardware and free any associated storage. The MSCP then destroys the transmit and receive OU tasks servicing this line, returns queues to the queue pool, frees the storage used for the source/sink active device list ND block, and updates the source/sink active device list to indicate that this line is not in use.

| For X.25, if another line on the IOP is active, the adjust buffer allocation message is sent to the IOM running that line.

Cancel

#MSNCNCL (an entry point in #MSNVONN) processes the modify ND cancel/request. This routine is used to suspend the reconnection and reuse of the communications link by setting the recovery resource/activation state in the ND object to cancel. If the ND represents a nonswitched line, the recovery resource/activation state is set to cancel in all attached CD objects and LUD objects to complete the cancel processing.

For all attached LUD objects, all queued pending activate resource Request I/O instructions present when this instruction is issued are returned to the machine-interface response queue.

Continue

#MSNCNTU (an entry point in #MSNVONN) processes the modify ND continue request. This routine is used to enable the reuse of the communications link by setting the recovery resource/activation status in the ND object to continue and sending the initialize line message to the SDLC/BSC/MTAM IOM to request initialization of the line adapter. If the ND represents a switched line with autoanswer and is in the enabled state, the connect-in message is sent to the SDLC/BSC/MTAM IOM. The connect-in message response is sent to the MSCP task input queue when the autoanswer function is complete.

Modify Controller Description (Asynchronous)

Vary-On

The synchronous MSCP routine for vary-on CD has completed its function when the SDLC/BSC/MTAM IOM is requested to poll a station for exchange identification information. When the SDLC/BSC/MTAM IOM receives the response to this request, it sends a message to the MSCP and the following operations are performed:

- A request-contact or request-activate PU message with the exchange identification or the system services control point identification information is received from the SDLC/BSC/MTAM IOM. #MSCDRQC is invoked to validate the exchange identification information and to perform the following:
 - For SDLC, create the station IOM task, send a vary-on CD message to the station IOM, and send a contact message to the SDLC IOM.
 - For BSC and MTAM, send a contact message to the BSC/MTAM IOM.
- Response to the contact, and for SDLC, the vary-on CD message must be received before further processing can occur. #MSCDCON and #MSCDVCD both maintain and check status flags in the source/sink active device list to monitor these messages. For all controllers except the 3274, when the message(s) have been received, the CD contact successful event is signaled to indicate that the vary-on is complete.

Note: For a 3274 Controller, a request I/O message is built to send an activate physical unit to the 3274 Controller. When the ACTPU response is received, the CD contact successful event is signaled to indicate that the vary-on is complete.

The request contact or request activate physical unit message response is returned to the queue indicated in the message. If any LUDs attached to this CD are in the vary-on pending state, a vary-on-LUD message is sent for each LUD attached to the CD. If the CD indicates a primary station, the attached LUDs remain in the vary-on-pending state until an activate LU SNA command is received from the host system.

- #MSLDVOR processes the response to the vary-on-LUD message. If required, this routine builds an activate-LU request I/O and sends it to the station IOM. The station IOM sends an activate-LU request to the device and receives the response. If the activate-LU request I/O is not required, the vary-on is completed and the LUD contact successful event is signaled.
- When a response to the activate LU is received, #MSLDALR is invoked to complete the vary-on and to signal the LUD contact successful event to indicate that the vary-on operation is complete.

Dial

The dial request causes an initialize-line message to be sent from the synchronous MSCP function to the SDLC/BSC/MTAM IOM. When the response to this message is received by the MSCP task, the following operations are performed.

- When the autodial operation completes, a connect-out response is sent to the MSCP task. #MSCDACR is invoked to send the exchange identification request to the SDLC/BSC/MTAM IOM.
- #MSCDXID processes the response to the exchange identification information. The SDLC/BSC/MTAM IOM also sends a request-contact or request-activate-physical unit message when it receives a response to the request exchange identification. When a request-contact or request-activate-physical unit message is received, processing resumes as described under *Vary-On* for asynchronous operations.

Enable—Switched, Autoanswer

When a switched autoanswer line is enabled, a connect-in message is sent to the SDLC/BSC/MTAM IOM to cause it to wait for an incoming call. When an incoming call occurs, it is automatically answered, the SDLC/BSC/MTAM IOM sends the connect-in response to the MSCP task, and the following operations are performed:

- #MSCDACR processes the response to the connect-in message. If a connection has been established, an exchange identification request is sent to the SDLC/BSC/MTAM IOM.
- #MSCDXID processes the response to the exchange identification information. After the exchange identification response is received, the SDLC/BSC/MTAM IOM sends a request contact message or a request activate PU message. When a request-contact or request-activate physical unit message is received, processing resumes as described under *Vary-On* for asynchronous operations.

Error Conditions

Should an error occur in one of the preceding asynchronous operations, cleanup procedures restore the changes made during the operation that failed. The following sequence is entered at the point determined by the amount of cleanup that is required:

- For SDLC, a vary-off CD is sent to the station IOM. #MSCDTSK processes the response to restore queues and pointers for the station IOM task, and a disconnect message is sent to the SDLC/BSC/MTAM IOM.
- #MSCDONE is invoked to process the response to the disconnect message.
- An abandon-connection message for a switched line is sent to the SDLC/BSC/MTAM IOM to reset the line connection.

- #MSCDCLN processes the response to the abandon connection. If the line is enabled for autoanswer, this module builds the appropriate line parameters and sends an initialize-line message that conditions the line for a retry operation. Otherwise, #MSCDCLN restores object status and linkage, and restores source/sink active device list indicators and linkage.
- #MSILRSP checks if the initialize request was due to an error. In this case, #MSILRSP restores object status and linkage, and restores source/sink active device list indicators and linkage.

Unsolicited Input

Data can arrive in the system from devices on the teleprocessing network with routing code indicating that it should be sent to the MSCP. This can be data on the system services control point (SSCP)-LU or SSCP-PU flow path. The station IOM builds an unsolicited-input message and sends it to the MSCP queue. The MSCP determines the reason for the unsolicited data and invokes a routine to handle it or signals an event if the data must be directed above the machine interface.

Work station have three sources of information that cause data to be sent to the MSCP:

- The System Request key. This key indicates that the operator wants to interrupt the current operation. This information is for program control only and the MSCP signals an event. The event-related data contains the information associated with the system request message.
- The Test Request key. When this key is pressed, the MSCP signals an event to the machine interface with data identifying the device that issued the request.
- Record maintenance statistics request. When the error log buffer is full, a message is generated and the error data is sent to MSCP. The MSCP task invokes a routine to format the error log message and header information, and sends it to the error log task. A response is also sent to the remote work station to indicate that the error data was received.

Request I/O

A Request I/O instruction can be executed indicating that this is an MSCP request I/O operation. The request I/O message is routed to the MSCP queue to be handled by the MSCP task. This request allows a user program to communicate with an SNA device via the SSCP-LU or SSCP-PU session. The Request I/O instruction is used to send a request or response to the device. The object pointer in the request I/O source/sink request points to an LUD or CD that indicates an LU or PU flow, respectively.

Asynchronous Message Handling

When #MSCPTSK receives a message from the input queue, it uses the function code to invoke the appropriate routine. The function codes and the routines invoked are shown in Figure 27-1. The paragraphs that follow the chart describe the functions of the message processors internal to the #MSCPTSK module.

Function Code	Message Name	Procedure Name	Module Name
021B	Request discontact	#MSINOPS	#MSSWMGD
0281	Inoperative station	#MSINOPS	#MSSWMGD
0282	Inoperative line	#MSINOPL	#MSSWMGD
0284	Request contact	#MSCDRQC	#MSSWMGA
0286	Request activate physical unit (secondary)	#MSCDRQC	#MSSWMGA
0287	Request activate logical unit (secondary)	#MSLDALU	#MSLDMSG
0288	Inoperative station (BSC)	#MSINOPS	#MSSWMGD
0289	Protocol violation (MTAM/BSC)	INOPERAT	#MSCPTSK
028C	Connect in request	#MSCDCNI	#MSSWMGA
028D	X.25 reconnect request	#MSRECON	#MSSWMGS
0290	Request de-activate logical unit (normal)	#MSDALUN	#MSSWMGD
0291	Request de-activate logical unit (abnormal)	#MSDALUA	#MSSWMGD
0292	Request de-activate physical unit (normal)	#MSDAPUN	#MSSWMGD
0293	Request de-activate physical unit (abnormal)	#MSINOPS	#MSSWMGD
0294	Request disconnect (normal)	#MSDISCN	#MSSWMGD
0295	Request disconnect (abnormal)	#MSINOPS	#MSSWMGD
0297	Last session unbound	#MSDALUA	#MSSWMGD
0298	First session bound	#MSLDALU	#MSLDMSG
1035	Vary-on LUD (continue) request	#MSLDVOC	#MSLDMSG
1042	MSCP asynchronous vary-on LUD request	#MSLDVOQ	#MSLDMSG
1043	MSCP asynchronous vary-off LUD request	#MSLDVFQ	#MSLDMSG
1047	MSCP asynchronous abandon call request	ABANCALL	#MSCPTSK
104F	Second time-out notification	#MSCDTOT	#MSSWMGD
2004	Unsolicited LU request	UNSOLINL	#MSCPTSK
2006	Unsolicited PU request	AELUNSPU	#MSCPTSK
2008	Unsolicited line bid (BSC)	INOPERAT	#MSCPTSK
2009	Suspended stream data received MTAM	INOPERAT	#MSCPTSK

Figure 27-1 (Part 1 of 2). Message Handling Routines

Function Code	Message Name	Procedure Name	Module Name
200A	Stream restored (MTAM)	INOPERAT	#MSCPTSK
200B	Incoming data discarded (MTAM)	INOPERAT	#MSCPTSK
200C	Device conflict (MTAM)	INOPERAT	#MSCPTSK
5002	Request I/O	#MSSRQIO	#MSSRQIO
5010	Asynchronous resource management request I/O	#MSSRQIO	#MSSRQIO
5020	Synchronous resource management request I/O	#MSSRQIO	#MSSRQIO
7001	Request I/O control (continue)	#MSSRQIO	#MSSRQIO
8201	Response to contact	#MSCDCON	#MSSWMGA
8202	Response to discontact	#MSCDONE	#MSSWMGD
820E	Response to connect out	#MSCDACR	#MSSWMGS
820F	Response to abandon connection	#MSCDCLN	#MSSWMGD
8215	Response to initialize line	#MSILRSP	#MSSWMGS
8216	Response to activate connect in	#MSCDACR	#MSSWMGS
8217	Response to de-activate connect in	#MSCDCIO	#MSSWMGD
8218	Response to de-activate connect out	#MSCDCIO	#MSSWMGD
8240	Response to request exchange identification	#MSCDXID	#MSSWMGS
8284	Response to request contact	#MSCDRSP	#MSSWMGA
8295	Response to request disconnect	#MSCDRSP	#MSSWMGA
9007	Response to reset	RSPRESET	#MSCPTSK
9030	Response to vary-on LUD (cold)	#MSLDVOR	#MSLDMSG
9031	Response to vary-on CD (cold)	#MSCDVCD	#MSSWMGA
9032	Response to vary-on LUD (ERP)	#MSLDVOR	#MSLDMSG
9033	Response to vary-on CD (ERP)	#MSCDVCD	#MSSWMGA
903E	Response to vary-off LUD (ERP)	#MSLDVFE	#MSLDMSG
903F	Response to vary-off CD (ERP)	#MSCDTSK	#MSSWMGD
9040	Response to vary-off LUD (cold)	#MSLDVFR	#MSLDMSG
9041	Response to vary-off CD (cold)	#MSCDTSK	#MSSWMGD
B003	Response to error log	#MSAELRS	#MSAELRS
D002	Response to request I/O	RSPREQIO	#MSCPTSK
D002	Response to activate physical unit	#MSCDAPR	#MSCPTSK

Figure 27-1 (Part 2 of 2). Message Handling Routines

Unsolicited-Physical-Unit Request

AELUNSPU processes the unsolicited-physical-unit request. If the message is an SNA request maintenance statistics, a record error log message is built with an error log record descriptor and an error log data record. If the length of the unsolicited-physical-unit message indicates that only common error data is present, #MSAELRS is invoked with a pointer to the error log message. Otherwise, the error log message is sent to the error log queue. Also, a request I/O response message is built and sent to the IOM queue indicated in the CD object.

If the unsolicited data is an SNA negative response, the CD source/sink active device list block is checked for an abandon call pending flag. If the abandon call flag is set, the vary-off CD message is sent to the station IOM and the discontact message is sent to the SDLC IOM.

If the unsolicited data is an SNA request disconnect command, then the inoperative station message is built and sent to the MSCP task. This causes a discontact message to be sent to the SDLC IOM and vary-off CD (ERP) to be sent to the station IOM.

5250 Information Display System Error Log Request I/O Response

AELRSPHD processes the 5250 information display system error log Request I/O response. This procedure checks the user key field in the message to see if the response is a request. Also, the feedback record is checked for any errors.

If neither of these conditions exist, the abandon call pending flag in the CD source/sink active device list block is checked. If the flag is set, a discontact message is sent to the SDLC IOM and a vary-off CD message is sent to the native IOM.

Inoperative Message

Messages received from the BSC/MTAM IOM invoke INOPERAT to signal BSC/MTAM events. The response bit in the message is then set and the message is sent to the response queue indicated in the message.

Abandon Call Request Messages

For disable ND, abandon call ND, or dial out CD messages, if the ND represents a switched line for autodial, the de-activate-connect-out message is sent to the SDLC/BSC/MTAM IOM. If the ND represents a switched line which is not for autodial, the de-activate-connect-in message is sent to the SDLC/BSC/MTAM IOM. If the vary-on CD message has been sent to the station IOM, the reset CD message must be sent to the station IOM.

For abandon connection CD messages, if the CD recovery resource/activation state is active, then the abandon call message is returned with a status indicating that the object state was invalid.

If the modify CD abandon connection function is performed after a normal disconnect occurs (CD recovery resource/activation state is normal pending), then the abandon connection message is sent to the SDLC/BSC/MTAM IOM.

If the CD state is active, LUDs active, and the recovery resource/activation state is continue, then the reset LUD message is sent to all attached LUDs. When the reset LUD response is received, the vary-off LUD (error recovery procedure) message is sent to the IOM.

If the CD state is active LUDs varied-on, the reset LUD message is sent to the IOM for all varied-on LUDs regardless of the recovery resource/activation state in the CD. The vary-off LUD (cold) message is sent to the IOM when the reset LUD response is received.

If the CD state is varied-on, the reset CD message is sent to the station IOM. When the reset CD response is received, the vary-off CD (cold) message is sent to the station IOM. If a BSC/MTAM CD is being varied off, the discontact message is sent to the BSC/MTAM IOM.

For vary-off CD messages, if the CD state is varied on, the reset CD message is sent to the station IOM. When the reset CD response is received, the vary-off CD (cold) message is sent to the station IOM. If a BSC/MTAM CD is being varied off, the discontact message is sent to the BSC/MTAM IOM.

If the CD state is varied-on-pending, the abandon call message is returned to the synchronous MSCP with a good status.

Unsolicited-Logical-Unit Request

UNSOLINL processes the unsolicited-logical-unit request. If the message is an SNA function manager data request, a supervisory services request event is signaled. The subtype (formatted or unformatted) depends on the format indicator in the request.

If the message is an SNA data flow control request for logical unit status, then one of two events is signaled. If the message indicates a device available status, the LUD contact successful event is signaled and the LUD recovery resource/activation state is set to continue or active; if the message indicates a device unavailable status, the LUD device not available event is signaled and the LUD recovery resource/activation state is set to normal pending or normal pending/activation pending.

All events have system pointers to the LUD. The supervisory services request event contains additional data from the SNA request. The device-failure event contains 2 bytes of logical unit status. A request I/O message is then built to send an SNA response to the IOM queue specified in the LUD source/sink active device list block. The response flag is set in the unsolicited messages and the message is then sent to the return queue specified in the message.

Reset Response

RSPRESET processes the reset response. If the type field in the message indicates a LUD is being reset and the activate logical unit SNA command was previously sent to the device, the deactivate logical unit SNA command is sent to the native IOM. For a BSC/LU1 CD or when an activate LU was not sent, the vary-off LUD message is sent to the IOM.

If the type field in the message indicates a CD is being reset and the CD indicates a 5250 work station, the MSCP routine #MSAELRB is invoked. #MSAELRB builds a request maintenance statistics request I/O and sends it to the IOM queue indicated in the CD object. For a 3270 Controller a deactivate physical unit request I/O message is built and sent to the IOM queue indicated in the CD object. For an LU1 or peer secondary station, the connection is terminated by the following functions:

- When a nonzero response is received for an LU1 or APPC secondary station, the discontact and vary-off CD messages are not sent until MSCP receives a disconnect normal request.
- When a zero response is received, a discontact message is sent to the SDLC IOM and a vary-off CD message is sent to the station IOM.

Request I/O Response

RSPREQIO processes the response to a Request I/O instruction. If the user key in the message indicates that it is a response to an unsolicited-input message generated by the MSCP, the storage occupied by both the message and the feedback record is released. If the user key in the message indicates that the message was generated by the CD physical unit services, #MSCDAPR is invoked for an activate physical unit response, routine RSPDCTPU is invoked for deactivate physical unit response, otherwise, routine AELRSPHD is invoked. This routine processes error log responses. If the user key in the messages indicates that the message was generated by the LUD logical unit services, #MSLDALR or #MSLDDLRL is invoked. #MSLDALR processes the activate-logical-unit response. #MSLDDLRL processes the de-activate-logical-unit response.

Deactivate Physical Unit Response

RSPDCTPU processes the response to the de-activate physical unit SNA command. A discontact message is sent to the SDLC IOM. A vary-off CD message is sent to the station IOM.

Physical Unit Services Routines

The modules #MSSWMGA, #MSSWMGD, and #MSSWMGS contain the routines that process the request and response messages involving physical unit services. All message handling routines check the flags that indicate if the synchronous MSCP function is attempting an operation that should stop asynchronous MSCP processing. The routines that process response messages also free the storage occupied by the messages.

Request Contact (Primary SDLC and BSC/MTAM/XIOM) and Request Activate PU (Secondary): #MSCDRQC processes the request contact and request activate physical unit messages. For a nonswitched line or a switched dial connection, the exchange information field or the system services control point identification field is compared with the exchange information or system services control point identification field in the CD(s) attached to the line. If the fields match, the following operations are performed:

- For SDLC, a station IOM task is created, a vary-on CD message is sent to the station IOM queue, and a contact message is sent to the SDLC IOM queue.
- For BSC or MTAM, a contact message is sent to the BSC/MTAM IOM.

For X.25, when the request contact or request activate physical unit messages are received by the MSCP from the XIOM, the MSCP validates the remote XID system services control point identification the same as an SDLC line.

If the fields do not match, the CD-contact unsuccessful event containing pointers to the CD and ND and the exchange identification information or system services control point identification information received is signaled.

For an SDLC switched-answer connection, the exchange identification information field or the system services control point identification information field in the message is compared as follows:

- For SDLC, the exchange identification information field or the system services control point identification information field in the message is compared with the list of switched CDs that are in the vary-on-pending state.

If a match is found, the ND candidate list in the corresponding CD is checked to determine if the CD contains a pointer to the ND for the line that sent the message. If the ND is in the candidate list, a station IOM task is created and messages are sent as described for a leased line or a switched dial connection. If the ND is not in the candidate list, the CD-contact-unsuccessful event with system pointers to the CD and ND and the exchange identification information or system service control point identification received is then signaled. If a CD with matching exchange information is not found on the candidate list, an ND exchange identification-failure event or a system service control point identification failure event containing a system pointer to the ND and the exchange identification information or system service control point identification is signaled.

For a BSC or MTAM switched-answer connection, the ND eligibility list is checked to determine if a pointer to a varied-on-pending CD is present for the line that sent the message. If the CD is in the eligibility list, the exchange identification information field in the message is compared with the list of exchange identification information contained in the CD. If a match is found, messages are sent as described for leased line or a switched dial connection. If a CD with matching exchange identification information is not found, an ND exchange identification failure event containing a system pointer to the ND and exchange identification information is signaled.

If a CD is found in the eligibility list, but the exchange identification information field does not match any of those in the CD list of exchange identifications, the CD contact-unsuccessful event containing system pointers to the CD and ND exchange identification information is signaled.

On a switched line, if either the CD contact unsuccessful or the ND exchange identification or the system service control point identification-failure event is signaled, an abandon connection message is sent to the SDLC/BSC/MTAM/XIOM IOM.

For X.25 permanent virtual circuits, the MSCP tries to find a vary-on-pending CD for a permanent virtual circuit attached to the ND and logical channel entry that the XID system services control point identifier was received on. If a CD is found, the contact message is sent to the XIOM and the vary on CD message is sent to the station IOM. If a CD is not found, the CD contact unsuccessful event is signaled with an appropriate reason code.

For X.25 switched virtual circuits, the request contact/request activate physical unit processing is not analogous to switched SDLC processing. When the request contact/request activate physical unit message is received on a switched line, the CD is already identified. All that is left to be done is to ensure that the XID system services control point identifier that came off the line matches the XID system services control point identifier in the CD. If the match is successful, the contact message is sent to the XIOM and the vary-on CD message is sent to the station IOM. If the match is not successful, the CD contact unsuccessful event is signaled with the appropriate reason code.

Response to Contact: #MSCDCON processes responses to contact messages. If the status field in the message is good and if the source/sink active device list for SDLC indicates that the vary-on-CD response was received without error, then SDLCCOMN is invoked to complete the contact operation for all controller types except 3274. For a 3274 controller type, a request I/O message is built to send activate physical unit message to the 3274 Controller.

If the status field in the message indicates an error, then SDLCCERR is invoked to recover from the contact failure.

Response to Discontact: #MSCDONE processes responses to discontact messages. This routine invokes ABANCOMN to complete the recovery.

Response to Connect In/Out: #MSCDACR processes responses to connect in/out messages. If the status field in the message is good, a request exchange identification or system services control point identification message is sent to the SDLC/BSC/MTAM IOM. If the status field indicates an error other than from the result of an abandon call processing, the abandon connection message is sent to the SDLC/BSC/MTAM IOM, and a CD contact unsuccessful event containing pointers to the CD and ND is signaled if it is a response to connect out.

For X.25, the response to the connect out message is received by the MSCP after an outgoing call request is processed. If the status indicates that the call request completed successfully, then the message includes the logical channel ID the connection was assigned to. The MSCP updates the link control entry associated with that link control identifier with the CD pointer at this time. Also, the MSCP sends the request XID message to the XIOM. If the status in the response indicates that the call request did not complete successfully, the CD contact unsuccessful event is signaled with an appropriate reason code. The connect-in response is the same as SDLC.

Response to Abandon Connection: #MSCDCLN processes responses to abandon connection messages. If the ND for the line specifies autoanswer mode and is not the result of abandon call processing, an initialize-line message is sent to the SDLC/BSC/MTAM IOM. Otherwise, the UNHOOK procedure is invoked to perform cleanup.

Response to Initialize Line: #MSILRSP processes responses to initialize-line messages. If the status field in the message is good, the activate-connect-in message is sent to the SDLC/BSC/MTAM IOM. If the status field in the message is not good, the returned status is saved, and the MSCP recovery failed flag is set in the ND source/sink active device list.

For X.25, the initialize line response is not received by the asynchronous MSCP task. However, the initialize line message is sent to the XIOM by synchronous MSCP modify ND continue code.

Response to De-activate Connect In/Out: #MSCDCIO processes responses to de-activate connect in/out messages. If the status field in the message is good, or if the ND object is not in inoperative pending state, or if an abandon call dial out request is pending, the internal procedure unhook is invoked. When the above conditions are not met, the abandon connection message is sent to the SDLC/BSC/MTAM IOM.

Response to Request Exchange Identification: #MSCDXID processes responses to exchange identification requests. This routine sets a flag to indicate that the message was received and then frees the storage occupied by the message.

Response to Vary-On CD: #MSCDVCD processes responses to vary-on CD messages. If the status field in the message indicates good status, and the contact-response message was received without error, SDLCCOMN is invoked to complete the vary-on-CD operation for all controller types except 3274. For a 3274 Controller type, a request I/O message is built to send activate physical unit message to the 3274 controller. If the status field indicates an error, SDLCCERR is invoked to recover from the error.

Response to Vary-Off CD (SDLC Only): #MSCDTSK processes responses to vary-off CD messages. If the CD is not attached to an ND (vary-off for work station), the response bit in the abandon call message is set, and the message is returned to the response queue specified in the message. Otherwise, ABANCOMN is invoked to complete recovery.

Response to Activate Physical Unit (SDLC Only): #MSCDAPR processes responses to activate physical unit. #MSCDAPR is invoked from RSPREQIO (a routine within #MSCPTSK). If the error summary field in the activate physical unit request I/O feedback record is normal, the activate physical unit response is positive, and the activate physical unit response unit is acceptable. An internal procedure, SDLCCOMN, is called to complete the vary on of the CD. Otherwise and internal procedure, SDLCCERR is called to start recovery after the failure. If the activate physical unit response is negative or the activate physical unit response unit is unacceptable, the CD contact unsuccessful event is signaled by SDLCCERR; otherwise, the activate physical unit was not successful because of a hardware problem and MSCP is notified via the inoperative line or inoperative station messages.

Complete Vary-On CD: SDLCCOMN completes the vary-on CD operation and is invoked by #MSCDCON and #MSCDVCD. If the LUD-varied-on count in the CD is greater than zero, the CD state is updated to CD-active/LUDs-varied on. If the count is not greater than zero, then the state is updated to CD varied-on. The CD-contact-successful event containing system pointers to the CD and ND is signaled. If any LUDs attached to the CD are in the vary-on-pending state, a vary-on LUD message is sent to the station IOM.

For X.25, SDLCCOMN is called by #MSCDCON (contact response), #MSCDVCD (vary-on CD response), and #MSCDAPR (activate physical unit response) to complete the asynchronous vary-on CD processing. The CD is updated to varied-on if no attached LUDs are varied on, or is changed to active/LUDs varied-on state if at least one attached LUD is varied on. The CD contact successful event is signaled with a system pointer to the ND and the CD. The logical link identifier is also included in the event data for X.25. The vary-on LUD message is sent to the station IOM for all attached LUDs in the vary-on-pending state. The request contact or request activate physical unit message is returned to the line IOM.

Switched CD/ND Cleanup: UNHOOK performs the cleanup operation and is invoked by #MSCDCIO, #MSCDCLN, #MSINOPL, ABANCOMN, or #MSCDTOT. The ND state is set to the enabled state, and the flags in the ND source/sink active device list block are cleared. If a CD source/sink active device list block is chained to the ND block, the CD block is removed from the ND block and enqueued to the CD pending queue with all flags reset. If the CD block has an LUD block chain, then the CD state is set to CD-varied-on-pending and LUDs-varied-on-pending state; otherwise the state is set to the CD-vary-on-pending state.

For X.25, the function performed by UNHOOK is analogous to the function it performs for SDLC and BSC. The CD pointer in the link control entry is cleared for a switched virtual circuit. Also, the ND pointer in the CD is cleared for a switched virtual circuit. The CD is changed to vary-on-pending and the ND is changed to vary-on, if it has no other active CDs attached.

Starts Recovery After Vary-On Failure: SDLCERR starts the recovery procedure after a vary-on failure occurs and is invoked by #MSCDCON, #MSCDVCD, and #MSCDAPU. The discontact message is sent to the SDLC/BSC/MTAM IOM and the vary-off CD message is sent to the station IOM to destroy the task. If the CD indicates a primary station, the request activate physical unit message is returned to the line IOM with a negative response; otherwise, the request contact message is returned to the SDLC/BSC/MTAM IOM with a negative response.

Completes Abandon Processing: ABANCOMN completes the abandon processing and is invoked by #MSCDONE, #MSINOPS, #MSDISCN, and #MSCDTSK. If leased lines are being used and the discontact/vary-off CD messages were sent as a result of receiving an abandon call message from the synchronous MSCP routine, UNHOOK is invoked to perform cleanup. If a switched line is being used, the abandon connection message is sent to the SDLC/BSC/MTAM IOM. The response bit in the abandon call message is turned on, and the message is returned to the response queue indicated in the message.

For X.25, the abandon processing routine is called when an inoperative station message is received by the MSCP task. Also, this routine is called when both the vary-off CD and discontact messages are received by the MSCP task during abandon call processing. For an X.25 ND, this routine calls the UNHOOK routine. The abandon connection message is never sent to the XIOM.

Inoperative Line Request: #MSINOPL processes inoperative line requests. The recovery resource/activation state in the ND is set to inoperative pending and the ND line failure (inoperative) event is signaled with a system pointer to the ND object, the inoperative status code, OU number, and time stamp of the message. For a switched line, the system pointer to the CD object is included if attached to the ND.

The response bit is set in the message, and the message is sent to the response queue indicated in the message.

In addition to the above processing, when a CD is attached to the ND, the CD recovery resource/activation state is set to inoperative pending, and the internal procedures, INOPCOMN and UNHOOK (if required), are called.

Inoperative Station Request: #MSINOPS processes inoperative station requests. The recovery resource/activation state in the CD is set to inoperative pending and the CD failure (inoperative) event is signaled with a system pointer to the CD object, the inoperative status code, OU number, and the time stamp of the message.

The response bit is set in the message, and the message is sent to the response queue indicated in the message.

The internal procedures, INOPCOMN and ABANCOMM (not for local work station), are called to complete the recovery.

De-activate Logical Unit Abnormal, Device Failure: #MSDALUA processes de-activate abnormal device failures. The following events may be signaled:

Message Received	Event Signaled
Request De-activate LU Abnormal	LUD Device Failure
Peer Device Failure (Status Code Hex 001 or 002)	LU-to-LU Termination
Peer Device Failure (Status code Hex 001 or 003)	LUD no sessions
Peer Device Failure (Status Code Hex 0004)	LUD Idle Sessions
Peer Device Failure (Status Code Hex 8nnn)	CD Unbound Intervention

For peer device failure (status codes hex n001 or n003, the LUD recovery resource/activation state is set to continue or activation pending.

For peer device failure (status code hex n002 or request de-activate LU abnormal), the LUD recovery resource/activation state is set to inoperative pending or inoperative pending/activation pending.

For peer device failure (status code hex n001, n002, or n003), the CD recovery resource/activation state is tested for active or activation pending status. If either of these CD bits are on, the CD chain of LUDs is checked for LUDs whose recovery resource/activation state is active or activation pending. As a result of this chain check, the CD recovery resource/activation state may remain the same or be changed to activation pending or continue.

For request de-activate logical unit abnormal, the CD recovery resource/activation status bits are not changed.

The response bit is set in the message, and the message is sent to the response queue indicated in the message.

De-activate Physical Unit Normal Request: #MSDAPUN processes de-activate physical unit normal requests. The CD failure (SSCP-to-PU session inactive) event is signaled with a system pointer to the CD object and a status code of hex 0001.

The CD block is enqueued to the CD pending queue in the source/sink active device list.

The response bit is set in the message, and the message is sent to the response queue indicated in the message.

De-activate Logical Unit Normal Request: #MSDALUN processes de-activate logical unit normal request. The LUD failure (SSCP-to-LU session inactive) event is signaled with a system pointer to the LUD object and a status code of hex 0001.

The LUD recovery resource/activation state is changed to normal pending. If there are any pending activate resource request I/Os outstanding, the LUD recovery resource/activation state is changed to normal pending/activation pending.

The response bit is set in the message, and the message is sent to the response queue indicated in the message.

Disconnect Normal Request: #MSDISCN processes disconnect normal requests if the ND is for switched answer and there is no CD attached, the ND recovery resource/activation state is set to continue and the ND connection unsuccessful (ND disconnect unsuccessful) event is signaled with a system pointer to the ND object.

For a nonswitched line or a switched dial connection, prior to the request contact/activate physical unit message being received, the ND and CD recovery resource/activation state bits are set to continue (for switched) and normal pending (for nonswitched). The CD contact unsuccessful event is signaled with a system pointer to the ND and CD objects. For a switched dial connection, the internal procedure, ABANCOMN, is called or for a leased MTAM/secondary station, the internal procedure INOPCOMN is called.

For all other conditions, after the CD contact successful event has been signaled, the CD loss of contact event with a system pointer to the CD object is signaled. The ND recovery resource/activation state is set to continue and the CD recovery resource/activation state is set to normal pending. The internal procedure, ABANCOMN, is called. For an LU1 or secondary station, a message is enqueued to the CD pending queue in the source/sink active device list until the connection is reestablished.

The response bit is set in the message, and the message is sent to the response queue indicated in the message.

Complete Inoperative Processing: INOPCOMN completes the inoperative processing by:

- Dequeuing the CD message from the CD pending queue in the source/sink active device list.
- For each varied-on LUD attached to the controller, copying the CD recovery resource/activation state to the LUD recovery resource/activation state.
- Dequeuing from the activate resource pending queue in the LUD, any immediate request I/Os present and calling #SSBLDFB to build a feedback record to be returned to the response queue.
- Completing the abandon call processing, if in progress prior to the receipt of the inoperative request, by sending a disconnect message to the SDLC/BSC/MTAM IOM and a vary-off CD message to the station IOM.
- For a leased MTAM or nonswitched primary connection, signaling the CD primary intervention event with a system pointer to the CD object whenever an activate resource request I/O is enqueued to the activate resource pending queue in the source/sink active device list.

Time-out Request Message From #SSSNDR: #MSCDTOT processes the time-out request message. A synchronous modify request failed to complete within the specified time limit. The time-out request is sent to the MSCP to attempt the completion of the function and if necessary destroy the failing IOM task. This is accomplished as follows:

- All outstanding messages, except request I/Os, have the function code changed to a response for ease of processing in the event the message is returned while processing the time-out request.
- If the station IOM or device IOM failed to respond to the message, the time-out kill message is sent to the IOM queue to destroy the message and a disconnect message is sent to the SLDC/BSC/MTAM IOM to complete the take-down sequence.
- If the line IOM failed to respond to the message, a time-out kill message is sent to the line IOM queue to destroy the message.
- If the line IOM tries to return a time-out kill message, there is an intentional function check to force the following:
 - Associated source/sink objects are marked partially damaged.
 - The user must vary off, then vary on, the objects to recover.
 - The VMC log is examined to determine why the instruction timed out.
- For requests outstanding for a LUD object, the time-out request message is sent to the station IOM.

Connect-In Request

For X.25, this message is received by the MSCP from the XIOM when a call request message is received by the System/38. The connect-in request message contains the ND address, the logical channel ID, and the call request message. When this message is received, the MSCP searches for a switched CD in vary-on-pending state and continue status with a remote network address parameter that matches the call request. If no such CD is found, the ND connection failure event (00E,04,06) is signaled with a reason code of 0100.

If a CD is found, the remote password parameter is checked against the password in the call request. If the password check fails, the ND connection failure event is signaled with a reason code of 0200.

If the passwords match and reverse charging was requested in the call request, the CD that was found is checked to ensure that reverse charging is allowed for calls received for this CD. If reverse charging was requested and the CD does not allow it, the CD contact unsuccessful event (004,04,04) is signaled.

If reverse charging is allowed, the protocol requested in the call request is checked against the protocol configured in the CD. If the protocol check passes, the connect-in request is returned to the XIOM with a status indicating that the call should be accepted. Also, the MSCP updates the logical link entry that is associated with the logical link ID that was chosen by the adapter.

If any of the above checks fail, the connect-in request message is returned to the XIOM with a status indicating that the call should be rejected.

X.25 Reconnect Request

This message is received by the MSCP when the adapter assigns an X.25 switched virtual circuit to a different logical link. The two logical link entries involved are updated to reflect the logical link change.

Asynchronous MSCP Routines for X.25

The use of MSCP asynchronous routines can be explained by outlining several operations that describe the sequence of events.

Vary-On CD

The synchronous MSCP routine for modify CD (vary-on) completes its function after it tells the XIOM to poll a station for ID exchange information. When the XIOM receives this information, a message is sent to the MSCP task that starts the asynchronous vary-on CD sequence.

Asynchronous Vary-On CD Sequence

Request contact or request activate physical unit is received from the XIOM with XID or system services control point information. Also included in this message is the ND address and the logical link ID. #MSCDRQC is called to validate the XID or system services control point information, create a station IOM task, send a vary-on CD message to the station IOM and a request contact message to the IOM.

The remainder of the asynchronous vary-on CD processing is the same for X.25 as for SDLC.

Dial Sequence for X.25

When a user program opens a file to a LUD attached to an X.25 switched CD, the MSCP signals the manual intervention event. If the CD is configured as a dial-out CD, CPF issues a modify CD (dial) instruction. When the modify CD (dial) instruction is issued, the synchronous MSCP searches the ND list for an available ND. When an ND is found, the connect-out message is sent to the XIOM. The abandon call message and the initialize line message is not sent; instead, the connect-out message is handled asynchronously by the MSCP task. The state of the CD and the ND are updated and the modify CD (dial) instruction is complete.

The test to determine whether an ND is available to dial out on is described below:

- The MSCP tests to see if any switched logical link entries are available
- If at least one logical link entry is available, the MSCP sends a message to the XIOM to get the current status of the logical links from the X.25 adapter.
- The synchronous MSCP waits for a response to this message.

- If the response to this message indicates that there is an outgoing logical link available, the MSCP sends the connect-out message to the XIOM.
- If either of the above two tests fail, the current ND is passed by and the next ND in the list is tested.
- If the ND list is exhausted, the resource not available exception is signaled.

Answer Sequence

For the X.25 answer sequence, the MSCP will not send a connect-in message to the XIOM. All incoming switched logical links are enabled to accept incoming calls at vary-on ND time. The MSCP is notified of an incoming call request via the connect-in request message. When the connect-in request message is received by the MSCP, verification is performed. If all checks are successful, the request XID message is sent to the XIOM.

Logical Unit Services Routines

#MSLDMSG contains the routines that process responses to messages that are requests for logical unit services. The routines that handle response messages, free the storage for the message.

Synchronous Vary-On LUD (Cold) Request: #MSLDVOQ processes synchronous vary-on LUD requests. A vary-on LUD message is sent to the IOM only if the state of the CD is vary-on or greater. For a CD that indicates a state of varied-on-pending, the CD recovery resource/activation state is copied to the LUD recovery resource/activation state. The response bit in the message is turned on and returned to the response queue indicated in the message.

Synchronous Vary On LUD (Continue) Request: #MSLDVOC processes synchronous vary-on LUD (continue) requests. A vary-on LUD (error recovery procedure or cold) message is sent to the IOM only if the state of the CD is varied-on or greater and the CD recovery resource/activation state is not inoperative pending, normal pending, or cancel. If the CD indicates a BSC station, a vary-on LUD (cold) message is sent. If the CD indicates LU1, peer, or work station, a vary-on LUD (error recovery procedure) message is sent.

For a CD that indicates a state of varied-on-pending, the LUD recovery resource/activation state is checked for activation pending state. If the LUD state is activation pending, and the CD is switched with a recovery resource/activation state of continue, the switched intervention event is signaled.

For a CD with a recovery resource/activation state of inoperative pending or cancel, an exception is signaled indicating this condition. The response bit in the vary-on LUD (continue) request is turned on and the message is returned to the response queue indicated in the message.

Response to Vary-On LUD (Cold or Error Recovery Procedure): #MSLDVOR processes responses to vary-on-LUD messages. If the status field in the message indicates good status, the session definition data in the LUD is checked to determine if an activate-LU is required. If activation is required, an activate-LU request message is built with the MSCP queue specified as the response queue. This message is then sent to the IOM queue specified in the LUD source/sink active device list block.

If activate-LU is not required, the LUD state is changed to varied-on if the LUD is currently in the vary-on-pending state. If this is the first LUD being varied-on, the CD state is changed to CD active/LUDs varied-on; or if the LUD is in a state greater than varied-on, the CD state is changed to CD active/LUDs active. If an activate resource request I/O is pending (not APPC), the activate resource request I/O is returned to the machine-interface response queue and the CD and LUD recovery resource/activation state bits are set to active. The LUD contact-successful event is signaled with a system pointer to the LUD. The MSCP vary-on request message (if present) is located, the response bit in the message is turned on, and returned to the response queue indicated in the message.

If the status field in the message indicates hex 8000, and the LUD is attached to an LU1 CD, an activate-LU request has not been received from the primary station. Vary-on LUD processing is suspended until an activate LU request is received from the primary station.

If the status field in the message is nonzero (for LU1 not equal to hex 8000), a vary-off LUD message is sent to the IOM queue indicated in the LUD source/sink device list block. If the vary-on LUD is not the result of a synchronous MSCP vary-on LUD request, the LUD contact-unsuccessful event is signaled with a system pointer to the LUD.

Response to Activate Logical Unit (SDLC Only): #MSLDALR processes responses to activate-logical-unit messages. #MSLDALR is invoked from RSPREQIO (a routine within #MSCPTSK). If the feedback record code, the response length, and the positive response indicator are all good, the LUD state is changed to varied-on if currently in the varied-on-pending state. If this is the first LUD being varied on, or if the LUD is in a state greater than varied on, the CD state is changed to CD active/LUDs active. If an activate resource Request I/O is pending (not APPC), the activate resource request I/O is returned to the machine-interface response queue and the CD and LUD recovery resource/activation state bits are set to active. If the bit in the response indicates that the LU is available, the LUD contact-successful event is signaled with a system pointer to the LUD. The MSCP vary-on request message (if present) is located, the response bit in the message is turned on and returned to the response queue indicated in the message.

If the activate-LU response is negative or the length of the response is bad, a vary-off LUD message is built and sent to the IOM queue indicated in the LUD source/sink active device list block. If the activate LU is not the result of a synchronous MSCP vary-on LUD request, the LUD contact-unsuccessful event is signaled with a system pointer to the LUD and the activate LU response data.

Response to De-activate LU: #MSLDDLRL processes responses to de-activate-logical units. A vary-off LUD (cold) message is sent to the IOM.

Synchronous Vary-Off LUD Request: #MSLDFVQ processes synchronous vary-off LUD requests. A reset LUD message is sent to the IOM to quiesce the MSCP-LU data flow only if the LUD is in the vary-on state. If the LUD is not in the vary-on state, or for a LUD attached to a BSC CD for which the MSCP recovery for the device has not completed (LUD contact successful event has not been signaled), the response bit in the vary-off LUD request message is set, and the message is returned to the response queue indicated in the message.

Responses to Vary-Off LUD: #MSLDFVR and #MSLDFVE processes responses to vary-off LUD messages. For a synchronous vary-on/off LUD request, the response bit in the message is turned on, and the message is returned to the response queue indicated in the message. For a synchronous abandon call request, the pending vary-off LUD message count is checked for zero. When the count reaches zero, and the CD represents an LU1, peer, or work station controller, a reset CD message is sent to the station IOM.

For a BSC/MTAM CD, the reset CD message is changed to a reset CD response and sent to the MSCP input queue.

Request Activate LU from Primary Station and First Session Bound: #MSLDA LU processes activate logical unit from primary station requests. If the LUD is in the varied-on-pending state, it is changed to the varied-on state.

If this is the first LUD being varied on, the CD state is changed to active and the LUD state to varied-on, or if the LUD is in a state greater than varied-on, the CD state is changed to CD active/LUD active. If an activate resource request I/O is pending (LU1 only), or the LUD state is greater than varied on (LU1 only) the activate resource request I/O is returned to the machine interface response queue, and the CD and LUD recovery resource/activation state bits are set to active. The LUD contact-successful event is signaled with a system pointer to the LUD. The response bit in the message is turned on and returned to the response queue indicated in the message.

If the LUD state is varied off or the recovery resource/activation state is inoperative-pending or cancel, the LUD contact-unsuccessful event is signaled (LU1 only) with a system pointer to the LUD. A good response code is inserted into the message, the response bit in the message is turned on, and the message is returned to the response queue indicated in the message.

When the first session is bound for a peer LUD, the CD and LUD recovery resource/activation state bits are set to active. However, if the LUD recovery resource/activation status is inoperative pending or cancel, a bad response code is inserted into the message, the response bit in the message is turned on, and the message is returned to the queue indicated in the message.

Error Log Routines

The error log routines support the request maintenance statistics, record maintenance statistics, and record error log message functions for the MSCP asynchronous task.

Response to Error Log: #MSAELRS processes responses to the error log. A pointer to the unsolicited-physical-unit request message is contained in the error log message. The response flag is set in the message, and the message is sent to the response queue specified in the message. The storage for the error log message is then released.

Build and Send Request Maintenance Statistics Request I/O: #MSAELRB processes the request maintenance statistics request I/O. Storage is allocated for a request maintenance statistics message. The request descriptor field is initialized and #MSREQB is invoked to build the request I/O message. The request maintenance statistics request information unit is then built in the message, and the message is sent to the IOM queue specified in the CD.

Logical Unit/Physical Unit Request I/O Routines

#MSSRQIO provides the support to handle the I/O requests issued by the process for the MSCP-LU and MSCP-PU sessions. The MSCP receives all requests but does not necessarily provide the function and may in turn route the I/O request to the associated IOM. This support is provided for the following MSCP request I/O functions:

- Default operations (SSCP-LU, SSCP-PU)
- Record formatted maintenance statistics type 00 alert operations (SSCP-PU)
- Return record formatted maintenance statistics type 00 alert operations (SSCP-PU)
- Activate resource (MSCP-LU)
- De-activate resource (MSCP-LU)
- Return activate resource (MSCP-LU, MSCP-PU)

Record Formatted Maintenance Statistics Alert Operations: CDREQIO or LUDREQIO processes the record formatted maintenance statistics request. The CD or LUD object state is checked for varied-on or greater. If the state is varied-on or greater, the I/O request is sent to the IOM queue indicated in the object.

Activate Resource Request: ARREQIO processes the activate resource I/O request. For a LUD attached to a BSC, LU1, or work station CD, the activate resource request I/O is never routed to an IOM. When the LUD is not contacted, the activate resource I/O request is enqueued to the activate resource queue in the LUD source/sink active device list. When LUD contact occurs or when an activate resource I/O request is received after LUD contact occurs, a feedback record for the activate resource I/O request is returned to the response queue.

For a LUD attached to a peer CD, the activate resource I/O request is handled by the station IOM. Before the LUD is varied-on, the LUD object state is varied on pending and the activate resource I/O request is enqueued to the activate resource queue in the LUD source/sink active device list. When the station IOM receives the vary-on LUD (cold) message from the MSCP, the station IOM takes the activate resource I/O requests off the activate resource queue in the LUD source/sink active device list and enqueues them on the LUDs mode table waiting queues. After the station IOM returns the vary-on LUD (cold) message, and until the LUD is varied off (cold), all activate resource I/O requests are routed to the station IOM.

When the LUD has not been varied on for the first time, or if it is recovering after a failure, one of the following occurs:

- For a switched line, when the recovery resource/activation state for the CD and LUD is continue, the CD switched intervention event is signaled.
- For an MTAM or primary peer nonswitched line, when the recovery resource/activation state for the CD and LUD are normal pending, the CD primary intervention event is signaled.

De-activate Resource Request: DRREQIO processes the de-activate resource I/O request. The activate resource, specified in the SSR, is dequeued from the activate resource queue in the LUD source/sink active device list or from a mode table waiting queue and returned to the machine-interface response queue.

For a LUD attached to a peer CD, the de-activate resource I/O request is routed to the station IOM; otherwise, it is returned to the machine-interface response queue with normal completion status.

Return Activate Resource Request: RAREQIO processes the return activate resource I/O request. All activate resource I/O requests are dequeued from the activate resource queue in the LUD source/sink active device list and returned to the response queue. When the system pointer in the SSR specifies a CD, this process is done for all varied-on LUDs attached to the CD.

For a peer CD, the return activate resource I/O request is routed to the station IOM; otherwise, it is returned to the machine-interface response queue with normal completion status.

BSC/MTAM AUTOMATIC RECOVERY TASK (BART)

The BSC/MTAM automatic recovery task (BART) is created for the BSC/MTAM I/O managers (IOMs) for processing line and station failures. BART is not directly associated with any specific communications line. When a line or station failure occurs, the associated IOM invokes #MSBERP and #MSBART to create BART. A BART is created for each BSC/MTAM IOM line or station failure. If multiple line or station failures occur, multiple BARTs are created.

The failing IOM invokes #MSBERP. #MSBERP first creates a new IOM queue, then #MSBART creates the BART. All information between the failing IOM and #MSBERP is passed through the BART control block. When BART is created, register 5 is pointing to the BART control block. Storage for the BART control block is controlled by the MSCP to prevent the storage from being freed if the IOM is destroyed before BART is destroyed. BART uses the IOM's old queue as the main send/receive queue.

DATA AREAS

Source/Sink Active Device List

The source/sink active device list is a data structure consisting of a fixed base with additional control blocks allocated as each ND, CD, or LUD is varied on. The ND, CD, and LUD blocks (not the same as ND, CD, and LUD objects) are chained to the base source/sink active device list. These control blocks contain information maintained and used by the various MSCP routines. The source/sink active device list base entries are as follows:

- **MSCP Task Input Queue Header (MSCPQ):** This queue header is used as an input queue for messages intended for the MSCP task.
- **MSCP Queue Pool Header (MSSQP):** Queues used by the MSCP functions are managed using a pooling scheme. Unused queues are placed on a queue pool queue header as send/receive messages. Access to the unused queues is via the module #MSFECHQ. When an MSCP function needs a queue, #MSFECHQ is invoked to get the next unused queue. When a queue is returned to the pool, the caller changes the descriptor byte to a send/receive message and enqueues the message to MSSQP.
- **MSCP Task Process Control BLock (PCB) Address (MSS@PCB):** The MSCP initialization module stores the current PCB in the source/sink active device list. All MSCP functions use this PCB address when getting or freeing machine-wide storage.
- **Directly-Attached CD Block Pointer (MSSCNPTR):** CD blocks for controllers that are directly attached (not associated with an ND) to the system are chained to this pointer.
- **Local LUD LU Block Pointer (MSSLUPTR):** LUD blocks for local LUDs (those with no CD or ND attached) are chained to this pointer.
- **Queue Pool Pointer (MSSQLPPT):** Blocks of queues in the queue pool are chained to this pointer.
- **Set Line Priority Parameters (MSSSETLP):** Each IOC has an 8-byte set line priorities field reserved in the source/sink active device list. These fields are completed at IOC wakeup and are used by the activate link function.

Active Line Flags (MSSAFLAG): Each IOC for communications has a 1-byte active line flag field in the source/sink active device list. Each bit represents a pair (0-8, 1-9, and so on) of lines on an IOC. A 0-bit indicates an inactive line; a 1-bit indicates an active line. These flags are used to determine if IOC wakeup should be performed.

Sender Identification Pool Pointer (MSSIDADR): A field of 4096 bits, chained to by this pointer, represents which sender identifications (0 through 4095) are used or are available for use. The assignment of the identifications is managed by #MSGETID (get an identification from the identification pool) and #MSRELID (return an identification to the identification pool).

ND Block Pointer Array (MSSNB@): The ND blocks are addressed through an array of thirty-two 6-byte pointers (MSSNB@); if an ND block does not exist, the pointer is 0. The first entry in the array is for OU#20, followed by 21 through 27, 60 through 67, A0 through A7, and E0 through E7.

CD Pending Queue (MSSCPQUE): When a CD attached via a switched network is varied on, the CD is set to the vary-on-pending state until the connection is made. Since the CD is not associated with a particular ND (and line) until the call is completed, the CD block is enqueued to the CD pending queue.

Active Line Speed Counters (MSSLVONC and MSSLVDMC): Each IOC for communications has a 2-byte counter in the source/sink active device list. Each counter represents the BPS values of the varied-on lines (1 = < 48 000 BPS, 4 = > 48 000 BPS). These counters are used to ensure that a high speed line is not varied on concurrently with any other line or lines that share the same IOC.

OU/ND Table Pointer (MSSPOUND): The OU/ND table is used to keep track of how many NDs exist (per OU) and which ones are varied on or in diagnostic state.

In addition to the fixed portion of the source/sink active device list, a block of data is generated and chained to the source/sink active device list whenever a CD, LUD, or ND is varied on. These blocks contain pointers to queues, task dispatching elements (TDEs), objects, related blocks, and status flags.

BART Control Block

The BART control block, shown in Figure 27-2, is the only control field used by BART. Because BART does not communicate with anything outside the system, the BART control block has no protocol characters or fields to be used.

BART Queue Pointer (old IOM queue)
IOM New Queue Pointer (not used by BART)
Failing ND Pointer
Failing CD Pointer
Count of Active LUDs
Feedback Record to be Returned
Saved Feedback Record
Device Dependent Code
Hardware Error Code
Error Time-Stamp
OU Number
Identification Of the Failing IOM
Status Flags
BART Control Table

Figure 27-2. BART Control Table

STRUCTURE

The following is a list of the MSCP modules and the function that each module performs. The list also shows how the module is invoked.

#MSAELRB Request Build

Function: Builds and sends the request I/O message to request maintenance statistics.

How Invoked: Within this component.

#MSAELRS Error Log Response Handler

Function: Sends the error log response to the response queue indicated in the message.

How Invoked: Within this component.

#MSBART BSC/MTAM Automatic Recovery Task

Function: Handles messages on the IOM's old queue.

How Invoked: Another VMC component.

#MSBERP BSC/MTAM Error Recovery Procedures

Function: Handles the creation of the new IOM queue and creates the #MSBART module needed by the BSC/MTAM IOM error recovery.

How Invoked: Another VMC component.

#MSCDIAL Dial On Switched Lines

Function: Initiates a manual or autodial operation to a switched controller.

How Invoked: Source/Sink instruction processor.

#MSCPTSK MSCP Asynchronous Task

Function: Initializes the source/sink active device list and processes MSCP requests.

How Invoked: System initialization; other VMC components.

#MSCVONN CD Status Control

Function: Modifies the status of the controller description as indicated by the source/sink instruction processor.

How Invoked: Source/Sink instruction processor.

#MSFECHQ MSCP Queue Management Subroutine

Function: Returns an MSCP queue element from a pool to the caller.

How Invoked: Within this component.

#MSGETID MSCP Station Identification Generator

Function: Searches the identification pool for the first unused identification and returns the identification to the caller.

How Invoked: Within this component.

#MSLDMSG Logical Unit Services

Function: Processes all messages pertaining to logical units (LUDs) that appear on the MSCP input queue.

How Invoked: Responses to messages from this component returned by IOMs.

#MSLSCRT Create/Destroy Session

Function: Activates and de-activates a session between the using process and the logical unit.

How Invoked: Source/Sink instruction processor.

#MSLVONN LUD Status Control

Function: Varies the status of the logical unit description as indicated by the instruction processor.

How Invoked: Source/Sink instruction processor.

#MSNVONN ND Status Control

Function: Varies the status of the network description as indicated by the instruction processor.

How Invoked: Source/Sink instruction processor.

#MSOUQCT Find and Initialize a QCT

Function: Finds and initializes a queue control table (QCT) for a given operational unit number.

How Invoked: Within this component.

#MSRELID Release MSCP Station Identification

Function: Returns an identification to the pool.

How Invoked: Within this component.

#MSREQB MSCP Build Request I/O Message

Function: Creates the Request I/O and feedback messages in machine-wide storage.

How Invoked: Within this component.

#MSSRQIO MSCP Request I/O Handler

Function: Handles all MSCP Request I/O instructions issued by the MI process.

How Invoked: Source/sink instruction processor.

#MSSWMGA Physical Unit Services #1

Function: Processes messages on the MSCP input queue that pertain to the asynchronous contact of physical units (controller descriptions).

How Invoked: Request messages from IOMs or responses from IOMs to request messages from this component.

#MSSWMGD Physical Unit Services #2

Function: Processes messages on the MSCP input queue that pertain to disconnecting physical units (controller descriptions).

How Invoked: Request messages from IOMs or responses from IOMs to request messages from this component.

#MSSWMGS Physical Unit Services #3

Function: Processes messages on the MSCP input queue that pertain to establishing connections of physical units (controller descriptions).

How Invoked: Request messages from IOMs or responses from IOMs to request messages from this component.

MULTI-LEAVING Telecommunications Access Method I/O Manager

INTRODUCTION

The MULTI-LEAVING telecommunications access method (MTAM) I/O manager (IOM) activates, manages, and de-activates the telecommunications channel and enforces the MTAM protocol. An MTAM I/O manager task is created by the machine services control point (MSCP) as a result of a Modify Network Description (vary-on) instruction. The MTAM I/O manager is used for communications to devices on a switched or nonswitched point-to-point telecommunications network. The basic MTAM support runs online using a subset of the BSC protocol commands.

An MTAM I/O manager interfaces with the following:

- The MSCP
- The error log
- An I/O controller
- Modify Network Description instruction
- Modify Controller Description instruction
- Modify Logical Unit Description instruction
- Request I/O instruction

An MTAM I/O manager task is associated with one communications IOC line position and provides specific support as shown in Figure 28-1.

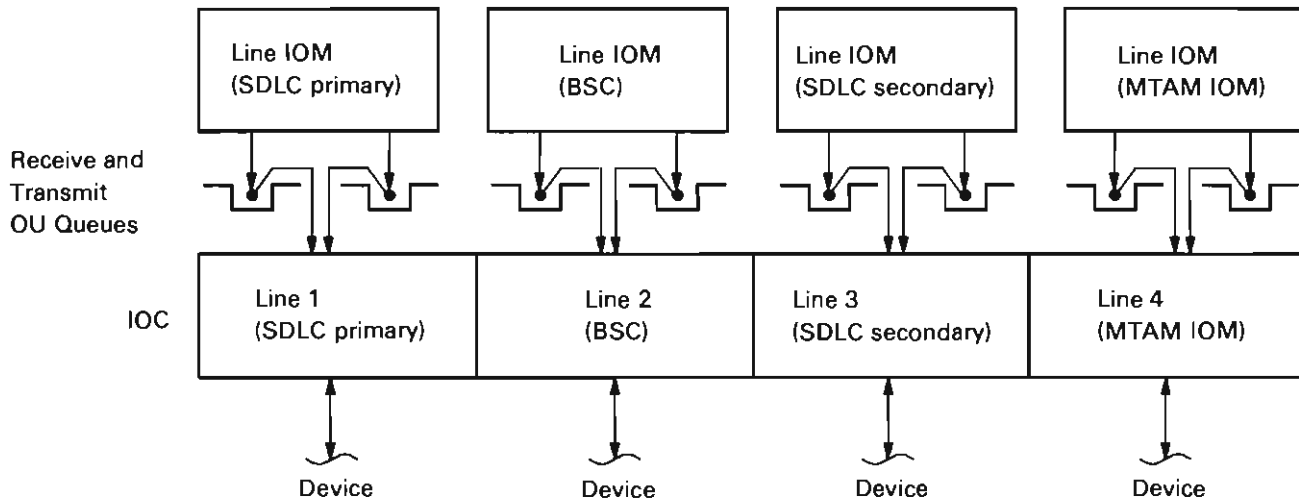


Figure 28-1. MULTI-LEAVING Telecommunications Access Method IOM/IOC Line Position Relationships

Communications to the MTAM IOM is through a send/receive queue as shown in Figure 28-2. The types of messages received are as follows:

- Source/sink instructions (from the CPF process)
- MSCP messages
- Operation request elements (OREs) (from either transmit or receive OUs)
- Time-out messages generated by the MTAM IOM (by the clock comparator task)
- Channel error messages (from the channel IOM)

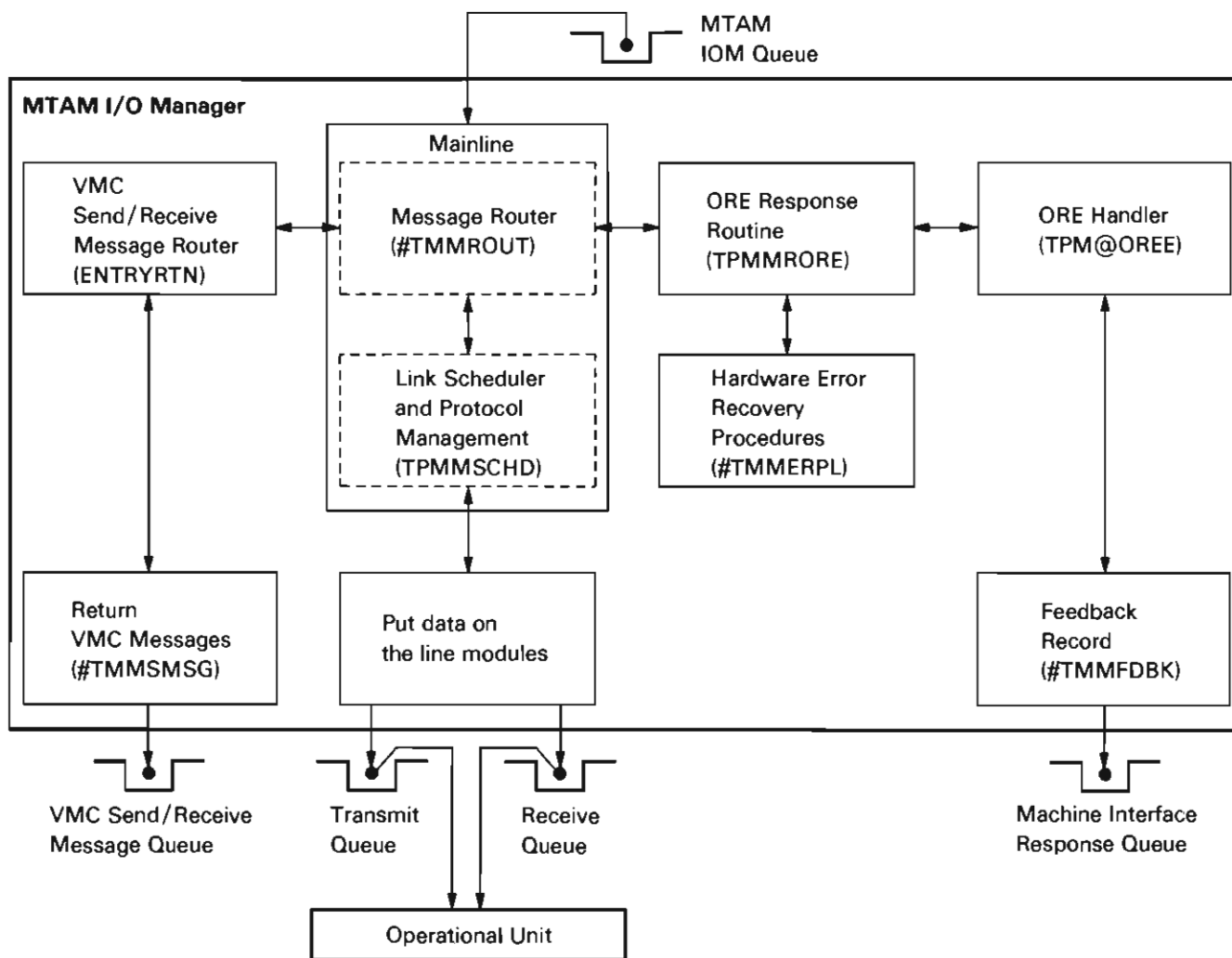


Figure 28-2. MTAM IOM

The queue message router receives the message and uses the key field in the message to determine if the message is a VMC message or an ORE. Then, based on the function field, the queue message router invokes the appropriate message (Figure 28-3) or ORE handler (Figure 28-4).

Function	Module
Request I/O	#TMMRQID
Request I/O (continue)	#TMMRQIC
Activate session	#TMMASES
Resume session	#TMMRSUM
Reset session	#TMMRSES
De-activate session	#TMMDSES
Initialize line	#TMMENB
De-activate connect-in	#TMMDABL
Establish switched connection	#TMMESC
Time-out	#TMMTIME
Vary-on LUD	#TMMVDNL
Dial (connect-out)	#TMMDIAL
Abandon connection	#TMMABO
Suspend session	#TMMSPND
Quiece session	#TMMQUES
Read data store	#TMMRDSO
Trap	#TMMTRPO
Request exchange identification	#TMMXID
Contact	#TMMCON
Discontact	#TMMDCON
Change ND retry values	#TMMNDRT
Message time-out	#TMMSFTD
Change device specific area	#TMMMDSA
Change line specific area	#TMMMLSA
Change user specific area	#TMMMUSC
Bad message	#TMMBADM

Figure 28-3. Message Function Handlers

Function	Module
Initialize line ORE	#TMMENBR
Reset line	#TMMDABR
Establish switched connection	#TMMESCR
Dial	#TMMDLR
Abandon connect-out	#TMMABDR
Activate link	#TMMVNNR
De-activate link	#TMMVOFR
Abandon connect-out line	#TMMABOR
Read data store	#TMMRDS2
Trap	#TMMTRP2
Request exchange identification	#TMMXIDR
Contact	#TMMCONR
Text processor	#TMMTXTR

Figure 28-4. ORE Function Handlers

To communicate with an MTAM host, the MTAM IOM issues write, read, and control type commands to the hardware/microcode support associated with the communications link attached to the IOC. See Figure 28-5. The IOC allows up to four communications lines. Two IOCs can be attached to the system so that the system can support eight communications lines. For synchronous transmission, a communications line may be activated as BSC point-to-point nonswitched or switched.

Prior to activation of any communications line IOC, the MTAM IOM through the MSCP verifies and ensures that the IOC is operational.

For each communications line on an IOC, there are two operational unit (OU) tasks, each with a send/receive queue. Two OU tasks per communications line allow a simultaneous transmit/receive capability useful for interrupting an operation by issuing a reset to the OU send/receive queue. These OU tasks and send/receive queues are created by the MSCP during line activation. The MTAM IOM maintains the send/receive queue addresses in the link control block.

Communication to the MTAM IOM is through a send/receive message called an operation request element (ORE). The operation block portion of the ORE is the portion where commands are specified, data areas are indicated, and status is returned. The MTAM IOM uses three types of operation blocks: the function operation block, the program operation block, and the message operation block.

The function operation block contains single commands such as initialize, establish switched connections, or line reset (line directed commands); as well as Write and Read commands.

The program operation block is used when multiple function operation blocks are to be executed. The program operation block references a chain of function operation blocks, each of which contains a command to be executed.

The message operation block is used during data transfer to eliminate the chance of command time-outs when two separate commands must be issued to the IOC for the execution of one I/O operation.

When the OU task has completed the activity requested by the command field of the function operation block, it sets the basic status (BSTAT) field of the function operation block. The first byte of the BSTAT (BSTAT0) is a general indication of the success of the operation. When BSTAT0 indicates an error, the second byte of the BSTAT (BSTAT1) gives a more specific indication of the cause of the error. The IOC returns the functional status (FSTAT) to the MTAM IOM during Read and Write operations.

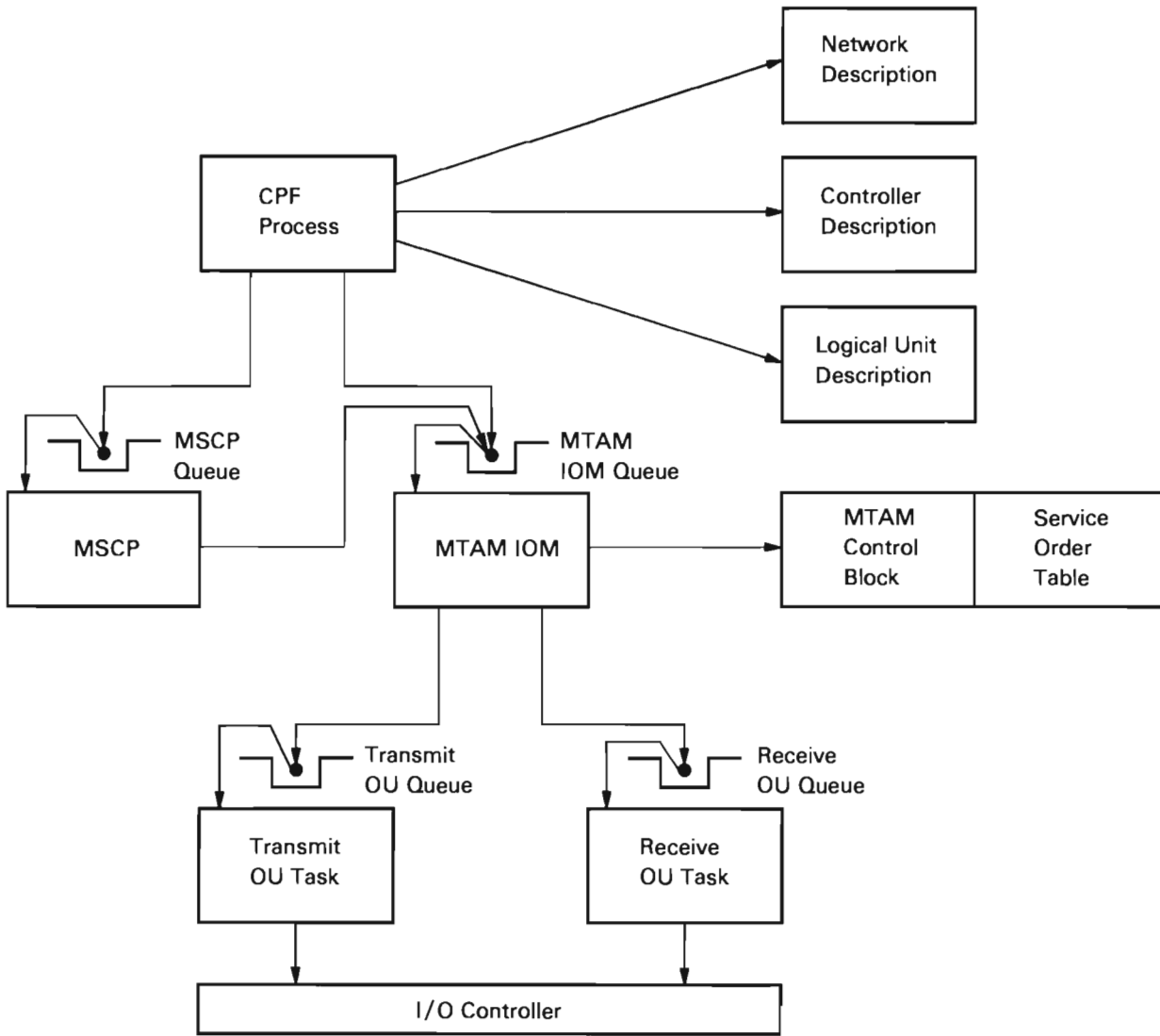


Figure 28-5. MTAM I/O Manager Interface

DATA AREAS

Link Control Block

The link control block (LKB) is in the invocation work area of the queue message router and is present when the task is active. The LKB contains the pointers, program operation block OREs, status indicators, and retry counters used for error recovery.

A pointer to the LKB is stored in register 4 and is saved throughout the task. The contents of the LKB is shown in Figure 28-6.

Program Operation Blocks
Operation Request Elements
Pointers
Sense Area
Retry Counters
Time-out Parameters
Service Order Table
Input Buffer
Alternate Input Buffer

Figure 28-6. Link Control Block

Service Order Table (SOT)

The service order table consists of 17 entries. Each entry represents a device. See Figure 28-7.

SOT Entry Number/Entry	
Input Devices	1 Console Keyboard
	2 Reader 1
	3 Reader 2
	4 Reader 3
	5 Reader 4
	6 Reader 5
	7 Reader 6
	8 Reader 7
Output Devices	9 Console Display
	10 Printer 1
	11 Printer 2 or Punch 7
	12 Printer 3 or Punch 6
	13 Printer 4 or Punch 5
	14 Printer 5 or Punch 4
	15 Printer 6 or Punch 3
	16 Printer 7 or Punch 2
	17 Punch 1

Figure 28-7. Service Order Table

Each SOT entry represents the device or devices indicated in the SOT and as shown in Figure 28-8, contains the following areas:

- Dequeue key
- Flags
- Pointers
- Data fields

<p>Dequeue Key</p> <ul style="list-style-type: none"> • Internal Queue Key (SOT Key)
<p>Flags</p> <ul style="list-style-type: none"> • Session Active Flag • Suspend Flag • Reset Flag • Quiesce Flag • System/38 BID Bit • Host BID Bit • Local Device Type • Host Device Type • Current Request I/O Valid Flag • End of File Found Flag • First Time Through Flag • Terminating Error Mode • Summary Flag
<p>Pointers</p> <ul style="list-style-type: none"> • Pointer to Internal Queue • Pointer to Logical Unit Description Object for Console, Reader, Printer • Pointer to Logical Unit Description Object for Punch Devices • Pointer to Modify Message • Pointer to Current Request I/O
<p>Data Fields</p> <ul style="list-style-type: none"> • Function Control Sequence Mask • Subrecord Control Byte Save Area • Trace Control Field

Figure 28-8. Service Order Table Entry

STRUCTURE

The following is a list of the modules in the MTAM IOM and the function that each module performs. This list also shows how the module is invoked.

#TPMELSE Nonmainline Functions/Protocol Error Recovery Procedure

Function: Handles the following routines:

- Start/cancel timer
- Read data store
- Trap
- Initialize
- Unsolicited data handler
- Request I/O (bid) instruction
- Error recovery procedures

How Invoked: Within this component.

#TMMERPL Horizontal Microcode Error Recovery and Logging

Function: Processes errors on the link as a result of horizontal microcode errors and does logging of the errors.

How Invoked: Within this component.

#TMMMIOM Mainline Function

Function: Handles the mainline functions of the MTAM IOM and routes all the incoming messages.

How Invoked: Other VMC component.

#TMMMODC Modify CD

Function: Processes messages that are sent to the MTAM IOM as a result of a Modify CD instruction.

How Invoked: Within this component.

#TMMMODL Modify LUD

Function: Processes messages that are sent to the MTAM IOM as a result of a Modify LUD instruction.

How Invoked: Within this component.

#TMMMODN Modify ND

Function: Processes messages that are sent to the MTAM IOM as a result of a Modify ND instruction.

How Invoked: Within this component.



Native I/O Manager

INTRODUCTION

The native I/O manager (IOM) is a VMC task for local work stations that interfaces with:

- Machine services control point (MSCP)
- Error log
- Modify Controller Description instruction processor
- Modify Logical Unit Description instruction processor
- Request I/O instruction processor
- Transmit and receive operational unit (OU) tasks
- Service function task

The user of the native IOM can execute Modify Controller Description and Modify Logical Unit Description instructions, and can make, break, and manage a systems network architecture (SNA) path to a logical unit (LU) attached to the work station controller. The Request I/O instruction is used to communicate with a logical unit. The native IOM handles the logical path for each LU attached to the work station controller; the native IOM is the multiplexing point for all communications to and from the work station controller.

The native IOM provides the user with the capability to communicate with locally attached 5250 devices using an interface that is compatible with remotely attached 5250 devices.

There is a native IOM task for each work station controller. Each IOM can handle from 1 to 32 logical units at a time.

The interface to the native IOM is through send/receive messages (SRMs) including I/O requests, operation request elements (OREs), and the feedback record. See the *Source/Sink Data Areas* in the *VMC Overview* section of this manual for a description of these areas.

The native IOM is a VMC task that is created by MSCP when a Modify Controller Description (vary-on) instruction is issued against a work station controller description object. The task is created with one input queue used to hold the SRMs and OREs. MSCP also creates transmit and receive OU tasks for the native IOM associated with the work station controller.

The operation of the native IOM is shown in Figure 29-1. The native IOM uses a router module to invoke the appropriate routine. The router gains control when the native IOM task is created (activate CD time). The router immediately executes a receive to the native IOM queue and waits for an SRM or ORE. Once a message is received, the router performs a table lookup to locate the routine that performs the requested function. If the function is not in the table and the message is not an ORE, the SRM is checked to see if it is a response. If the SRM is a response, the storage occupied by the message is freed. (This message was generated by the native IOM and no further processing is required.) If the requested function is not supported by the native IOM, the message is returned with an unsupported status indicated. When the SRM or ORE has been processed, the router again does a receive on the native IOM queue and the process is repeated.

There is a module for each SRM function code and ORE command code.

There are three types of native IOM modules:

- System/38 instruction processor modules
- SNA support modules
- I/O support modules

The System/38 instruction processor modules provide direct support of the source/sink modify instruction processors. The SNA support modules are part of the input and output data path. The SNA support is used with the Request I/O instruction because the native IOM provides the interface that is compatible with remotely attached 5250 devices. The I/O support modules provide direct support of the Request I/O instruction (this support includes the SNA modules), and the OREs used to perform the I/O operations to the work station controller.

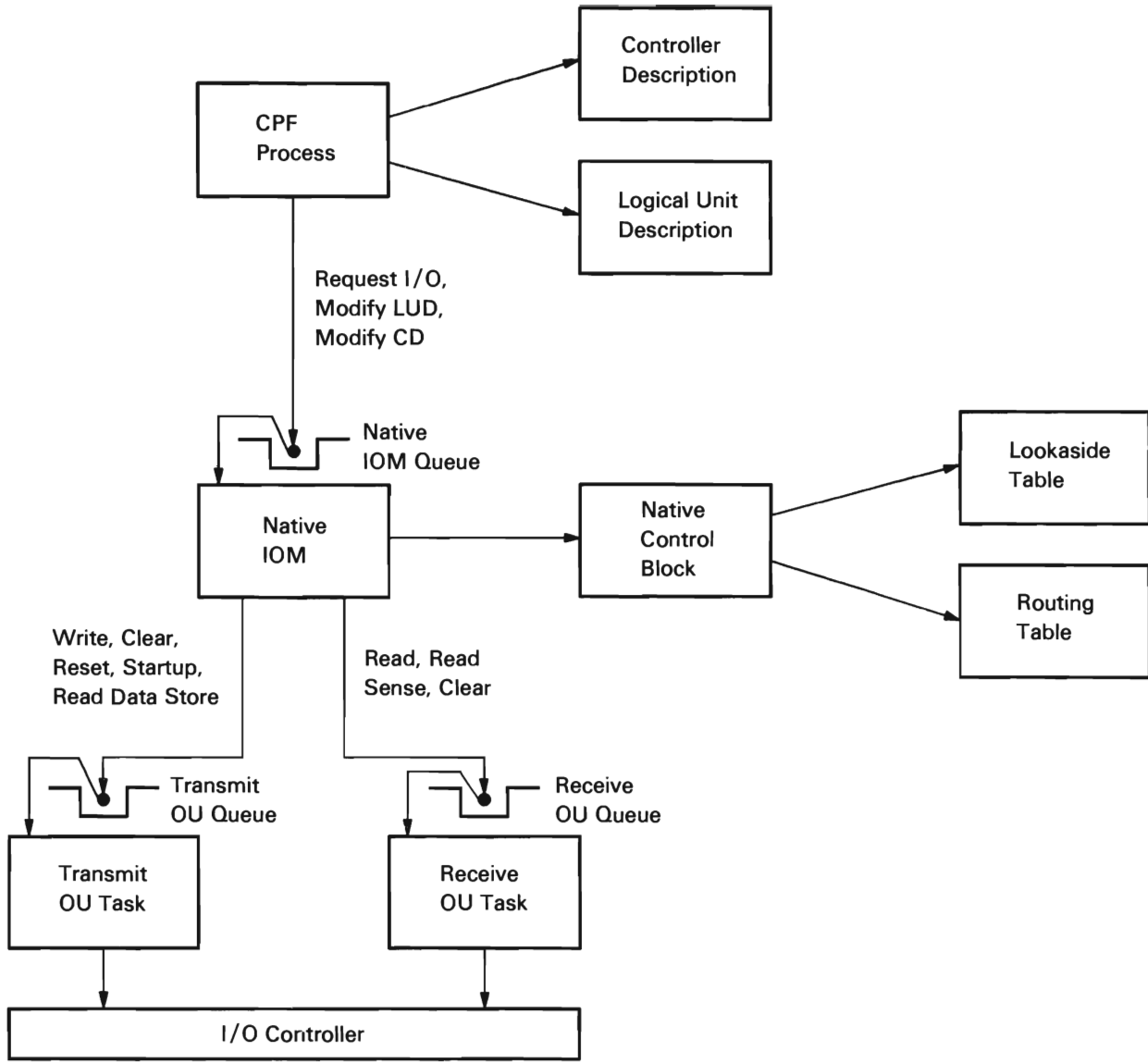


Figure 29-1. Native IOM Overview

System/38 Instruction Support

All source/sink modify instructions are passed to the native IOM in the form of messages (SRMs). These messages are built by mapping the instructions to the VMC format. This step is performed by the source/sink instruction processors.

All messages are routed to the appropriate modules by the native IOM router. The instructions supported are described in the following paragraphs.

Modify Controller Description (Vary-On/Off): This instruction is used to establish or break the communications path to the physical unit in the station represented by a controller description.

Modify Controller Description (Cancel): This instruction is used to suspend the reconnection and reuse of the controller description (CD). An SRM is sent to the native IOM causing the CD to go into error recovery mode and suspend request I/O processing.

Modify Controller Description (Continue): This instruction is used to allow the reuse of the CD and request that contact be established after an unrecoverable error occurs. An SRM is sent to the native IOM causing the CD to reset error recovery mode after an inoperative station failure and allow processing to continue.

Modify Controller Description (Unit-Specific Contents): This instruction is used to modify the unit-specific contents of the CD. The native IOM does not use any of the modifiable unit-specific contents in its CD.

Modify Logical Unit Description (Vary-On/Off): This instruction is used to establish or break the communications path from the MSCP to the logical unit. It also performs preliminary setup for a Modify Logical Unit Description (activate) instruction to establish a logical unit to logical unit path.

Modify Logical Unit Description (Cancel): This instruction is used to suspend use of the device after an unrecoverable error. An SRM is sent to the native IOM. The native IOM uses the object pointer in the SRM to locate the proper entry in the routing table and puts the device represented by the LUD listed in the routing table into error recovery mode.

Modify Logical Unit Description (Continue): This instruction is used to allow the reuse of the device after an unrecoverable error occurs. An SRM is sent to the native IOM activating the MSCP-PU control path.

Modify Logical Unit Description (Activate/De-activate): This instruction is used to establish or break the communications path from the system logical unit to a work station controller logical unit.

Resume: A Modify Logical Unit Description (activate) instruction is altered to a resume command by the source/sink instruction processors when the logical unit is in the inactive state (quiesce, suspend, reset).

Modify Logical Unit Description (Quiesce): Once the native IOM receives this request, it completes all I/O requests that are either in process or waiting on the queue. When processing of this instruction is completed, device management reaches a normal completion and all I/O request processing is stopped until a Modify Logical Unit Description (activate) instruction is issued.

Modify Logical Unit Description (Reset): This instruction causes all I/O requests to be returned to the user request queue, with an indicator that the request was not processed. The number of request descriptors processed is also set in the feedback record. This instruction also places the path in an inactive state that can be reactivated with a Modify Logical Unit Description (activate) instruction. All available unsolicited data is purged by this instruction.

Modify Logical Unit Description (Suspend): This instruction ensures that all I/O requests for logical units are in the suspend state. This means that if all I/O requests have been started, those that contain transmit requests only or receive requests only will be completed, and those that contain transmit/receive requests will have only the transmit portion completed. The I/O requests that complete are put on the return queue with the appropriate feedback code and the number of request descriptors (RDs) processed. Any transmit/receive type of I/O request with only the transmit portion complete remains on the queue of the logical unit.

Modify Logical Unit Description (Device-Specific Contents): This instruction is used to modify the device-specific contents of the logical unit description (LUD). Since the CD is already varied on, the native IOM is used to verify the device-specific contents of the LUD whenever the device-specific contents are modified. The native IOM then returns either positive or negative status to the user. These modifications will not take effect until a Modify Controller Description (vary-off) and a Modify Controller Description (vary-on) instruction is issued.

Request I/O: This instruction is used for two purposes:

- Perform I/O operations on the various paths (MSCP to physical unit, MSCP to logical unit, logical unit to logical unit).
- Return the logical unit to an active path state after a terminating error has occurred (Request I/O-continue).

The I/O function is described under *I/O Support* in this chapter. The Request I/O (continue) instruction uses a termination error mode bit in the routing table. This bit is set when a terminating error is encountered. The user is informed of the error by a code in the feedback record. If the user can recover from the error, the user resets the mode bit with the Request I/O (continue) instruction, and normal I/O processing resumes.

SNA Support

The native IOM provides a locally attached interface that is compatible with remotely attached 5250 devices. The native IOM uses SNA support equivalent to the remote station IOM and the remote work station controller. This support consists of the data flow control and transmission subsystem layers of SNA 0081(T3). The data flow control layer is that of a first speaker (bind receiver—similar to a secondary station arrangement). The transmission subsystem layer is contained entirely within the native IOM, including both host and terminal modules. The transmission subsystem provides the orderly exchange of data between logical units. It consists of path control and connection point manager.

Path Control

Path control uses the information in the T3 transmission header to control delivery of received basic information units to the addresses, supervisory services, or LUs in a node (including a clustered group of LUs).

Path control is broken into the following areas:

- Routing
- Function identification translation
- Logical I/O paths

The routing function consists of a search of the lookaside table to find the correct route table entry. In the outbound direction, the search depends on the LU being varied on. This information is obtained from the Request I/O instruction. This information along with the requested I/O function (MSCP or normal) is used to locate the correct path (MSCP-PU, MSCP-LU, or LU-LU). In the inbound direction, the destination address from the transmission header is used to locate the correct logical unit in the routing and lookaside tables. The transmission header flow bits are then used to point to the LU-LU or MSCP-LU path (the transmission header is in the read data ORE). For either inbound or outbound traffic, once the correct route table and lookaside table entries are located, the native IOM performs the necessary steps to deliver the basic information unit.

Function identification translation converts the internal transmission header and request/response header to the work station controller write-ORE parameters, and from work station controller read-ORE parameters to the internal transmission header and request/response header. This process provides the common interface for both remotely and locally attached 5250 devices.

Because the native IOM is performing this conversion from function manager parameters to the internal transmission header and request/response header and vice versa, it is not necessary for the native IOM to perform many of the transmission header and request/response header format checks.

Once the translation and routing functions have located the correct path (for example, routing table and lookaside table entries), the native IOM selects the correct logical I/O path for the frame. The frame or I/O request contains the data necessary to identify the correct logical I/O path. An I/O path is selected by setting the basing variable. SNA defines the logical I/O paths as either expedited or normal for both transmission and receive. These logical I/O paths are represented in each routing table entry.

Connection Point Manager

The connection point manager is the control point within the logical unit for distribution of the request units to the appropriate LU component (session control, data flow control, and function manager data). It also enforces pacing for those logical units that require pacing.

Connection point manager routes all session control request units whenever they are received, but routes data flow control (DFC) and function management request units only when the session is active. This function is done only for outbound traffic. The distribution for inbound traffic is done by the user process based on data in the feedback record and receive request descriptors.

Pacing is a means whereby the receiving connection point manager can control the rate at which it receives requests on the normal data flow. Responses are not paced. Also, expedited data is not paced.

Pacing is only needed for printers attached to the work station controller. Pacing is only used for output printer data, and is initiated by the pacing count in the logical unit description.

Each routing table entry has fields defined to be used for pacing a path. The fields are the pacing count and counter of frames sent. If the pacing count is 0, pacing is not invoked.

Pacing is started by setting the pacing count to the pace limit and the counter to 0. This initial state specifies that the first frame is to be sent and the counter to be incremented by one. Subsequent frames are sent out until the counter equals the pace count (limit). The pacing response from the LU causes the counter to be decremented by the pacing count. This means that the terminal can handle N more frames (N being the pace limit). The LU controls the flow by sending the pacing response when it is ready to receive more frames.

The native IOM and work station controller pacing differs from SNA 0081 pacing as follows:

- The pacing response is not requested from the work station controller. The response is passed to the native IOM when the work station controller has enough (pacing count) buffers available.
- The pacing state is set to the initial state when a component-aborted situation is encountered Request Shutdown command.

Session Control

Session control commands are used to initiate and terminate sessions for end-to-end communication between LUs. Session control T3 allows only four commands (activate LU, de-activate LU, bind, and unbind). Activate and de-activate LU are used by the MSCP to establish the MSCP-LU session. Bind and unbind are used by the function manager (host) to establish the LU-LU session and exchange LU descriptions and security data. The native IOM does not maintain any session control state machines. It only translates the SNA commands to the work station controller write data parameter list and the read data response parameters, and these parameters to SNA command responses.

The native IOM uses the LUD specific characteristics area to build the bind parameter list for the work station controller.

Data Flow Control

The native IOM uses the DFC first speaker (bind receiver). This means that the native IOM does not have any responsibility for recovery; when an error is encountered, the native IOM enters a receive state and waits for orders from the host.

DFC monitors and controls the exchange of requests and responses to and from the function manager according to the half-duplex send/receive mode of operation. DFC controls the flow of request units to the function manager by processing the request/response header bits which specify chaining, response type, direction, and flow.

Chaining is the process whereby DFC groups the request units for recovery purposes. All request units passing through DFC use chaining. The native IOM generates the correct chaining bits for inbound data (data from the work station controller).

The native IOM enforces the following rules regarding response type and direction:

- A request/response header indicating end-chain requires a definite response (with or without change direction) or an exception response with CD.
- All request/response headers indicating the start or middle chain and not the end chain cannot request change direction, and must request an exception response.

DFC processes two flows for all request units:

- Data flow control
- Function manager data

The request/response header of the function manager data request units is monitored for managing DFC-state machines. The native IOM generates the outbound DFC command request units and decodes the inbound DFC commands. The DFC commands exist only as parameters in the work station controller read-data ORE. The native IOM converts those parameters to the actual SNA request units before the requests are passed to the host in the feedback record.

The DFC commands are:

- Cancel: This command is used to terminate the chain currently being sent. This command is executed either because the sending LU detected an error before the complete chain was sent or a negative response to the chain was received.

The native IOM generates this DFC command only if the work station controller returns a read data ORE with the Cancel command set on. The work station controller sets the Cancel command on if the work station controller lost contact with the device during a read operation.

When the Cancel command is received, the native IOM builds a null write function operation block (FOB) (FOB with zero length request/response unit), and sends it to the work station controller. The Cancel command also turns on the cancel in progress bit in the FOB. The null write FOB causes the work station controller to clear the SNA purging chain state and return a positive response.

- Logical Unit Status command (SNA): This command is used to report the status of a device while the device is in the send state. The work station controller uses this command to report component failures and component availability conditions.
- Request Shutdown command (SNA): This command indicates that this LU is ready to be de-activated. Since the host of the native IOM is the bind sender, the host is required to send an unbind command. The native IOM returns a negative response (function not supported) when it receives a Request Shutdown command (SNA) from the host. The work station controller generates the Request Shutdown command (SNA) whenever it encounters a component failure. The work station controller turns on a Request Shutdown command (SNA) flag in a read-data ORE, indicating negative response or the Logical Unit Status command (SNA). The native IOM maps the read-data ORE into the negative response or the Logical Unit Status command (SNA), followed by the Request Shutdown command (SNA). The native IOM also causes the pacing counter to be set to 0 and the Logical Unit Status command (SNA) component abort to be sent to MSCP.
- Signal: This command is used for both inbound and outbound operations on the expedited flow, and is used to request or report attention, request a direction change, turn on or off a message indicator, and indicate manual intervention. The native IOM does special handling for attention and request change direction. The native IOM ensures that only one attention is outstanding at one time. Any additional attentions are purged.

The signal request change direction is supported through the work station controller write-data ORE hold-read option. When the work station controller receives this request, it ensures that any LU read currently in progress is not allowed to complete. The LU read is suspended until a subsequent LU data stream command (a restore or data-stream read of any type) releases the read. When a write-data ORE completes, the native IOM returns the CD to the host.

The signal request change direction is sent to the native IOM as a conditional request. This means that the native IOM must first ensure that a LU read is outstanding before processing the request. If a read is not outstanding, the request change direction request I/O is returned not processed.

MSCP Functions

The native IOM provides the MSCP with an interface similar to the one provided to the logical unit. Two differences apply to this interface:

- The MSCP-LU or MSCP-PU path is not placed in terminating-error mode. Thus, a continue is not required to resume request I/O operations.
- If a suspend command is issued, the native IOM assumes that no additional Request I/O instructions will be issued for that path until a resume request is issued.

The MSCP either processes unsolicited data or passes the unsolicited data to the user as data in events. The events are either formatted or unformatted supervisory services data. The MSCP always returns the unsolicited data SRM with a successful completion code.

I/O Support

Output

The output process is started by a request I/O spooling function (#TPNRQIO). This function receives all I/O requests from the native IOM queue and enqueues the requests on the correct LU queue. The requests are always enqueued last on this queue and are categorized according to the I/O path on which they are to be executed (for example, expedited and normal transmit, expedited and normal receive, receive any, receive immediate). The spooler uses the second byte of the SRM key for these encodings.

Once the requests are moved to the appropriate LU queue, the scheduler is invoked to process the output. The scheduler consists of two parts, a loop selection and a logical I/O path selection. The loop selection uses one lookaside table entry as the basic unit. This module ensures that each LU gets an equal opportunity for output and ensures that each transmit ORE contains data for only one LU.

The logical I/O path selection module (build station output) gets control from the loop selection when the next lookaside table entry is selected. Addressability to the route table entry is established before the logical I/O path is selected. This section locates the next request to be processed (logical I/O path), enforcing the path control rule that expedited processing is executed before normal processing. The selection routine also controls the building of the work station controller write data FOBs. For Kanji devices, module #TPEIDO (for display output) or module #TPEIPO (for printer output) is called to translate ideographic extension characters into RAM addresses and to load the device ideographic RAM.

The correct path is selected from the transmit path message pointers in the routing table entry. If the pointer is 0, the appropriate 2-byte key is built and a dequeue equal is done on the LU queue. If a request I/O is not located, the process returns to the loop selection module and the next LU is processed. If an I/O request is located, the output FOBs are built for that request if required.

When the FOBs are built, the SNA finite state machines are updated and the transmission header, request/response header, are translated into the work station controller FOB command format.

On return, the loop scheduler determines whether it is time to build an operational program (any FOBs to be sent). If not, the loop scheduler selects the next entry and repeats the selection sequence. If the operational program is to be built, the loop scheduler builds the program, sends it to the transmit OU, and puts the transmit OU in a busy status.

Output Posting

The native IOM now waits for the output to complete or input to arrive. This can occur in any order, depending on whether or not the output operation was successful.

If the output operation was successful, the operational program returns control to the native IOM prior to any input resulting from the output, and the write-date ORE response handling module does the following:

1. If the output was for a display or printer (printer when end-chain not sent), the module marks the transmit RDs as being processed. This routine then builds a feedback record when either all RDs are processed, or all transmits are processed and a receive RD is processed with an end-chain indicator on.
2. If the output is for a printer and end-chain is indicated, and the work station controller is in a purging-chain-state (negative response to a chain occurred), the write-data response module processes the request as in the preceding step.
3. If the output is for a printer and end-chain is indicated, and the work station controller is not in a purging-chain-state, the write data response does not mark the RDs as being processed. The RDs are marked as being processed when the read-data ORE with the work station controller requested response bit set is received.

If the output operation was not successful, the output response can be received after the input operation (negative response for this chain). If the output response is received first, the write-data response module does not mark the RDs as being processed. This is done by the read-data-response module when it receives the negative response. This applies to both printers and displays.

Input

The native IOM does not explicitly request for a specific logical unit input from the work station controller. The work station controller polls the logical units for input and, in effect, the native IOM has a read outstanding. The work station controller passes the input to the native IOM through a read-data ORE. This ORE contains the location, length, flow, and the logical unit address for the request unit. The input processor (#TPNLUCL) uses this information to perform the translation from the ORE/function operation block (FOB) format of the work station controller to the transmission header and request/response header format of SNA. The input processor also identifies the data flow control module that is to process the request unit. This data is stored in the native control block for use by the request I/O-locate and DFC modules. For Kanji displays, module #TPEIDI is called to re-translate RAM addresses in the data stream back into ideographic extension characters and to handle ideographic alternative entry, LUSTAT. The input processor then passes control to the request I/O-locate module.

The request I/O-locate module selects the correct logical I/O path on which the frame is to be sent. If an I/O request is not pending or no buffer space is available, the frame is considered unsolicited data and an event or feedback record respectively is returned to the user. If the native IOM is in the receive state, the frame is also considered unsolicited data but the user is not informed of this occurrence. This is done by the native IOM because the work station controller passes any available input to the native IOM and because the work station controller is not aware of SNA send/receive states.

Unsolicited data is held on the queue of the logical unit in the form of an ORE. This ORE is built by the native IOM and has the same format as the read-data ORE. The ORE also contains a data area for a copy of the data stored in the native control block by the input processor and the request unit. This allows the native IOM to free the input buffer even if the user has not supplied a buffer for the data. Because the unsolicited data is formatted as a read-data ORE, it is processed by the same modules used for input. The routing table entry contains bits that indicate that the scheduler is to process this data.

If a request I/O is located, the DFC module that is to process this request unit is invoked. DFC finite state machines are updated at this point, causing SNA 0081 data flow control protocols to be enforced. Once this is completed, the connection point manager send-operation posts the request/response unit to be returned to the user using the request I/O. The native IOM then waits for the next message to be placed on the input queue.

DATA AREAS

Also refer to *Source/Sink Data Areas* in the *Vertical Microcode Overview* section of this manual for additional description of source/sink data areas.

Controller Description

The CD is used to identify the OU number of the work station controller and the chain of logical units attached to the work station controller.

The input/output controller (IOC) wakeup procedure uses the chain of LUDs in order to construct the parameters for the work station controller Load-Poll-List command. This command establishes the polling sequence, LU addressing, and LU translate table.

The native IOM uses the CD at activate CD time. The native IOM saves the OU number in the native control blocks and obtains the machine-wide storage needed to build internal control blocks.

Logical Unit Description

The LUD is used to describe the device type, logical session identifier, cable and head address, translate table, format table length and features attached. At vary-on LUD time, the native IOM uses this control block to establish the routing table data needed to communicate with the LU.

Native Control Block

Figure 29-2 shows an overview of the native control block. The native control block is built in the routing module (#TPNNIOM) in the invocation work area at activate controller description time. The native control block is a common control point for managing the work station controller. The native control block contains a subset of the data in the CD and is used to avoid page faults on user-owned CD objects. The native control block is always in main storage when the native IOM task is executing.

The native control block provides a directory function to other data areas stored in the machine-wide storage. The native control block also functions as a collection point for native characteristics, control point for native output, and common location for unique SNA and native work areas.

Pointers

- Controller Description
- LU Queue
- Routing Table
- User Buffer
- Input Buffers
- FOB Chain
- Lookaside Table

Output Control

- Status Flags
- LU Service Index
- Counters

SNA and Work Area

- SNA Sense Code
- Transmission Header
- Request/Response Header
- Message Counter
- Destination Address Field Index Table

Figure 29-2. Native Control Block

Routing Table

The routing table is allocated and initialized in machine-wide storage at activate CD time. The routing table supports the SNA transmission subsystem. This table contains fields that support path control (routing and expedited/normal I/O paths), connection point manager (pacing), data flow control, and logical unit (I/O queueing and session states).

An entry in the routing table represents an SNA path. Figure 29-3 shows an overview of a routing table entry. There are two entries per logical unit attached to a work station controller, one for each of the LU-LU and MSCP-LU sessions. These entries are established at LUD vary-on time. The MSCP-LU path is activated and ready for I/O traffic at that time. The LU-LU session is activated and ready for I/O traffic at modify LUD (activate) time.

The second entry in the routing table represents the MSCP path (MSCP-PU) for the work station controller. This path is activated and ready for I/O traffic at activate CD time.

Path Control

- Object
- Origin/Destination Address
- Transmit Flow
 - Expedited
 - Normal
- Receive Flow
 - Expedited
 - Normal
 - Immediate

Connection Point Manager

- Pacing Flags
- Pacing Counters

Logical Unit

- I/O Queue
- Session State Flags
- Unsolicited Data Flags
- Active Request I/O Counter

Figure 29-3. Routing Table Entry

Lookaside Table

The lookaside table, like the routing table, is allocated and initialized in machine-wide storage at activate CD time. The lookaside table contains additional fields to support the logical unit. An entry in the lookaside table also represents an SNA path and is the basic loop scheduling unit. A one-to-one correspondence exists between the entries of the lookaside table and the entries of the routing table. When scheduling output, a reference to the proper routing table entry is also established at this time. Figure 29-4 shows an overview of a lookaside table entry.

Logical Unit
• Transmit Request I/O Count
• Additional Session State Flags
• Additional Unsolicited Data Flags

Figure 29-4. Lookaside Table Entry

Operation Request Element and Program Operation Block

The ORE and the program operation block exist in the invocation work area of the native IOM, and exit during the execution of the IOM task. The ORE and the program operation block are used to request the OU task to execute a chain of FOBs.

Function Operation Block

The FOB is allocated in machine-wide storage. The native IOM allocates 32 FOBs.

Source/Sink Data Areas

These data areas belong to the Request I/O instruction user. They contain device-specific commands and output text. The data contained in the source/sink data areas is not referenced by the native IOM unless the data concerns data flow or session control, but is used by the OU task as the output buffers. However, if the source/sink data area is destroyed, either the OU task or the native IOM task will terminate.

STRUCTURE

The following is a list of the native IOM modules and the function that each module performs. The list also shows how the module is invoked.

#TPNCMDS Group Native IOM Commands

Function: Groups native IOM MSCP commands into one module. This module contains the following entry points:

- #TPNACCD: Vary-on CD
- #TPNVONL: Vary-on LUD
- #TPNACSS: Activate session
- #TPNDCSS: De-activate session
- #TPNVOFC: Vary-off CD
- #TPNVOFL: Vary-off LUD

How Invoked: Within this component.

#TPNEXCP Native IOM Exception Handler

Function: This module handles all exceptions in the native IOM. This module contains the following entry point:

- #TPNOUEX: Exception handler for OU task exceptions

How Invoked: Within this component and from the third level exception handler.

#TPNFUNC Group Native IOM Functions

Function: Groups native IOM functions into one module. This module contains the following entry points:

- **#TPNPECE:** Post event or channel error
- **#TPNIOER:** I/O error handler
- **#TPNDARR:** Work station controller reset response
- **#TPNDACR:** Work station controller clear response
- **#TPNRSR:** Work station controller read sense response
- **#TPNERLG:** Error log

How Invoked: Within this component.

#TPNIOCI Work Station Controller IOC Initialize

Function: Puts the work station controller IOC into normal mode.

How Invoked: Within this component.

#TPNLUCL Logical Unit Control and Function Manager Data

Function: Analyzes the read-data ORE response and converts it to the correct SNA transmission header, request/response header, and request unit.

How Invoked: Within this component.

#TPNNIOM Native IOM Queue Message Router

Function: This module is the main entry point in the native IOM. Routes SRMs and OREs to the appropriate module.

How Invoked: Other VMC components.

#TPNOPER Group Native IOM Operation Routines

Function: Groups operation routines into one module. This module contains the following entry points:

- **#TPNHDSM:** Handle read-data store SRM
- **#TPNBDSO:** Build read-data store ORE
- **#TPNHDSR:** Handle read-data store ORE response
- **#TPNSRTO:** SRM time-out handler

How Invoked: Within this component.

#TPNRIOL Request I/O Locate

Function: This module locates the proper I/O request to receive the input data.

How Invoked: Within this component.

#TPNRQIO Request I/O

Function: Receives all Request I/O instructions from the native IOM queue and enqueues the requests on the routing table with the correct key. This module contains the following entry point:

- **#TPNFDBK:** Request I/O feedback

How Invoked: Within this component.

#TPNSCED LU Output Scheduler

Function: Selects the routing table entry to be processed based on the loop (equal priority) scheduling and loops through the routing table until all entries have been checked or processed.

How Invoked: Within this component.

#TPNSERV Group Native IOM Service Routines

Function: Groups native IOM service routines into one module. This module contains the following entry points:

- #TPNQUSC: Quiesce
- #TPNRSET: Reset
- #TPNRSUM: Resume
- #TPNCONT: Continue
- #TPNPGLU: Purge LU
- #TPNSPND: Suspend
- #TPNUSOL: Unsolicited response
- #TPNPGUD: Purge unsolicited data

How Invoked: Within this component.

#TPNWDRS Write Data Response

Function: Processes all responses to write-data OREs. This module processes write-data ORE completions and I/O exceptions but invokes the I/O error handler for I/O errors and rejected commands.

How Invoked: Within this component.

#TPEIDO Ideographic Display Output

Function: Scans Kanji data, translates to RAM addresses, and manages ideographic RAM contents list.

How Invoked: Within this component.

#TPEIDI Ideographic Display Input

Function: Scans Kanji input data, translates RAM addresses into ideographic extension characters, processes alternative entry LUSTAT.

How Invoked: Within this component.

#TPEIPO Ideographic Printer Output

Function: Scans Kanji data, translates to RAM addresses, and manages ideographic RAM contents list.

How Invoked: Within this component.

#TPERCLM RAM Contents List Manager

Function: This set of subroutines performs common operations on the RAM contents list of ideographic devices.

How Invoked: Within this component.

#TPERROR Kanji Error Handler

Function: Handles errors unique to Kanji devices.

How Invoked: Within this component.



Secondary Station I/O Managers

INTRODUCTION

The secondary station IOM provides the function that allows System/38 logical units to communicate in an SNA network or in a system services control point (SSCP) network.

An LUC 6.2 LUD is active under the jurisdiction of the advanced program-to-program communications (APPC) station I/O manager. The secondary station IOM represents a secondary physical unit type 2 (PU.T2) controlling the secondary LU1, LU2, or LU3 LUDs.

The secondary physical unit type 2 (PU.T2) support consists of two VMC I/O managers (IOMs) and a common group of systems network architecture (SNA) modules. The secondary station IOM is the interface between the machine interface and the secondary line IOM or the X.25 line IOM. The secondary line IOM performs the synchronous data link control (SDLC) function. The common group of SNA modules (#NA2xxxx) performs path control, session control, physical unit (PU) and logical unit (LU) services, and connection point manager functions of the secondary station IOM.

The secondary station IOM interfaces with the following:

- Machine services control point (MSCP)
- Secondary line IOM
- X.25 line IOM (XIOM)
- Error log
- Modify Controller Description instruction processor
- Modify Logical Unit Description instruction processor
- Request I/O instruction processor

The user of the secondary station IOM can execute Modify Controller Description and Modify Logical Unit Description instructions, and can make, break, and manage an SNA path to an LU within the primary unit. The Request I/O instruction is used to communicate with an LU. The secondary station IOM handles the logical path for each LU within the primary unit to which the System/38 is connected.

An overview of the secondary IOMs is shown in Figure 30-1.

The secondary station IOM is a VMC task that is created by MSCP when a Modify Controller Description (vary-on) instruction is issued against a secondary controller description (CD) object. The task is created with one input send/receive queue upon which the send/receive messages are placed. The MSCP also provides the secondary station IOM with the address of the input send/receive queue of the secondary SDLC IOM and with the address of a controller description block for exception handling.

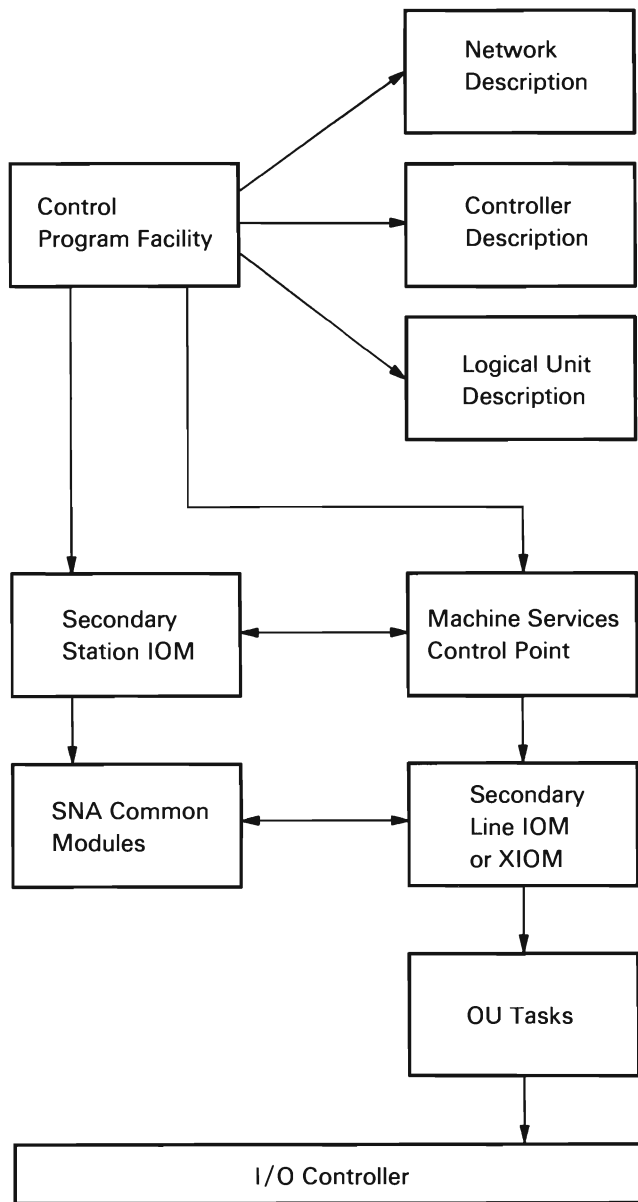


Figure 30-1. Secondary IOM Overview

The secondary station IOM uses a router module to invoke the appropriate routine. The router gains control when the secondary station IOM task is created (vary-on CD time). The router immediately executes a Receive Message instruction from the secondary station send/receive queue and waits for a send/receive message (SRM). Once an SRM is received, the router performs an if/then/else scan to invoke the routine that processes the requested function. If the function is not identified in the if/then/else scan, the SRM is checked to see if it is a response. If the SRM is a response, the storage occupied by the message is freed. (This message was generated by the secondary station IOM, and no further processing is required.) If the requested function is not supported by the secondary station IOM, the message is returned with an unsupported status indicated. When the SRM has been processed, the router executes a Receive Message instruction from the secondary station IOM send/receive queue, and the process is repeated.

Because there are unique function codes in the SRMs, the secondary station IOM has one unique callable routine per function and response. A routine is defined for a response only if further processing is required upon the return.

Following are the types of secondary station IOM modules:

- System/38 instruction processor modules
- SNA support modules
- I/O support modules
- DHCF (Distributed Host Command Facility) support modules

The System/38 instruction processor modules provide direct support for System/38 instructions. These modules recognize the requested function and initiate the processing of that function.

The SNA support modules are part of the output and input data paths. The I/O support consists of modules, including the SNA modules, that are in direct support of the Request I/O instruction.

The DHCF modules support the connection of a System/370 terminal to a System/38 application. Included within the DHCF support is 327x to 525x and 525x to 327x data stream translation. See Figure 30-2 for an overview of the secondary station IOM.

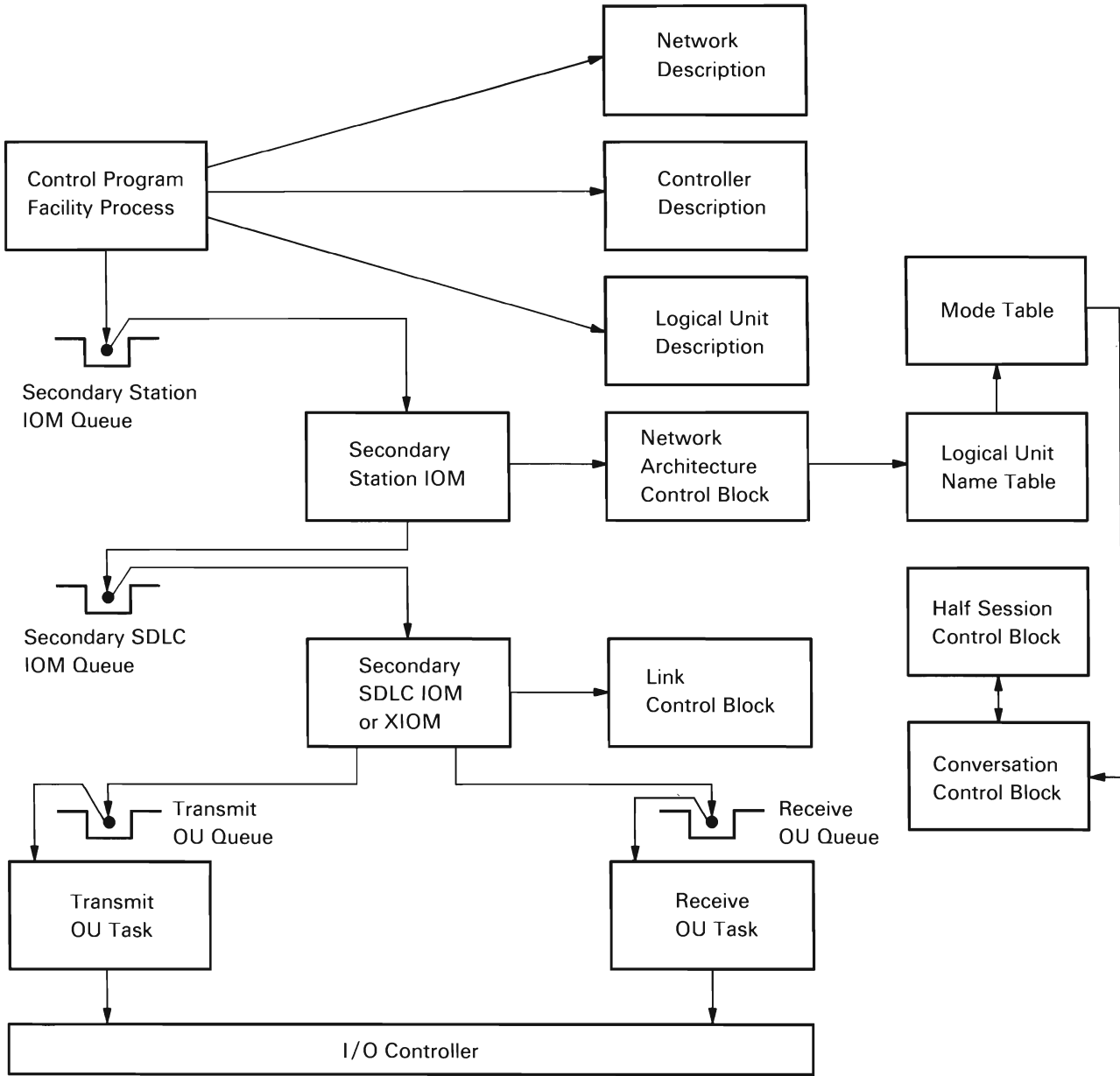


Figure 30-2. Secondary IOM Processing

System/38 Instruction Support

All source/sink modify instructions passed to the secondary IOMs are in the form of messages (SRMs). These messages are built by mapping the instructions to the VMC format. This step is performed by the source/sink instruction processors.

All messages are routed to the appropriate modules by the secondary IOM routers. The instructions supported are described in the following paragraphs.

Modify Controller Description (Vary-On/Off): This instruction is used by device management to establish or to break the communication path to the system services control point (SSCP) in the primary unit or station represented by the controller description from the physical unit (PU) in the secondary System/38.

Modify Controller Description (Dial): This instruction is used to allow a station to be dialed manually or automatically.

Modify Controller Description (Abandon Connection): This instruction is used to disconnect the secondary SDLC IOM from a remote station. The secondary SDLC IOM stays active; but the switched line is disconnected.

Modify Controller Description (Continue/Cancel): These instructions allow or suspend the reuse of the device after an unrecoverable error.

Modify Logical Unit Description (Activate/De-activate): This instruction is used to establish or to break the communications path from the System/38 LU to the remote (primary) LU.

A Modify Logical Unit Description (activate) instruction is converted to a resume command by the source/sink instruction processor when the LU is in an inactive state (quiesce, reset, suspend).

Modify Logical Unit Description (Vary-On/Off): This instruction is used to establish or to break the communications path from the SSCP to the System/38 secondary LU.

Modify Logical Unit Description (Quiesce): This instruction is used to complete all I/O requests that are either in process or waiting on the secondary station IOM's input queue. When processing of this instruction is complete, device management reaches a normal completion, and I/O request processing is stopped until a Modify Logical Unit Description (activate) instruction is issued.

Modify Logical Unit Description (Reset): This instruction causes each I/O request to be returned to the user return queue with an indicator that the request was not processed. The number of request descriptors (RDs) processed is set in the feedback record, and all available unsolicited data is destroyed. The communications path is placed in an inactive state that can be reactivated with a Modify Logical Unit Description (activate) instruction.

Modify Logical Unit Description (Suspend): This instruction ensures that all I/O requests are in a suspended state. All I/O requests that have been started are completed and returned to the user return queue with the appropriate feedback code and number of RDs processed. Processing of the suspended I/O requests can be resumed with a Modify Logical Unit (activate) instruction.

Modify Logical Unit Description (Continue/Cancel): This instruction allows or suspends the reuse of the device after an unrecoverable error.

Request I/O: This instruction:

- Performs I/O operations on the various paths: SSCP-PU, SSCP-LU, LU-LU
- Returns the LU to an active path state after a terminating error has occurred: Request I/O (continue)

The request I/O function is described under *I/O Support* later in this chapter.

The request I/O (continue) function uses a terminating error mode bit in the conversation control block. This bit is set on when a terminating error is encountered. The user is informed of the error by a code in the feedback record. If the user can recover from the error, the user resets the mode bit with the Request I/O (continue) instruction, and normal I/O processing resumes.

The following list is a brief description of the Request I/Os:

- *Request I/O (Activate Resource)*. This request activates the resource (LUD) for use by the requesting MI user.
- *Request I/O (De-activate Resource)*. This request sends pending activate resource request I/Os back to the user's return queue.
- *Request I/O (Return Activate Resources)*. This request sends back to the user's return queue all pending activate resource request I/Os associated with the LUD specified in the source/sink request with feedback record status indicating the condition.
- *Request I/O (Return Formatted Maintenance Statistics 00 Alert Operations)*. This request provides SSCP-PU flow alert support when communicating with a host that has system services control point. Alerts are not supported for connections to another System/38 or APPC peer devices.
- *Request I/O (Return Record Formatted Maintenance Statistics 00 Alert Request I/O instructions)*. This request sends back to the user's return queue all Record Maintenance Statistics 00 Alert request I/O instructions associated with the CD specified in the source/sink request.

- *Request I/O (Return REQIO)*. This request causes all outstanding request I/Os for this LUD to be sent back to the user's return queue with status indicating that a return request I/O instruction was executed.
- *Request I/O (Set PCS Address)*. This request causes the PCS address of the sending process to be saved by the secondary station IOM. This enables the secondary station IOM to signal the LU-LU normal flow unsolicited data event on a process-dependent basis rather than a machine-wide basis. The event continues to be signaled until a modify LUD (Reset) instruction is received.

SNA Support

The secondary SDLC IOM supports the data link control layer of SNA. The path control and transmission control layers are supported by the common SNA modules that are invoked by the secondary station IOM. The following features are supported:

- One PU, PU type 2 (PU.T2)
- Transmission subsystem profile 3
- Function manager profile 3
- Unformatted system services using the SNA character string for the presentation class on the SSCP-LU flow
- A unique SSCP identification for each line connected to the System/38

Path Control

Path control routes basic information units between the remote primary and the System/38 secondary half sessions so that the node/link configuration of the network is transparent to the half sessions. For System/38 inbound data, path control uses information in the format identification transmission header to control delivery of the basic information unit to either the specified supervisory services or an LU in a node (including a clustered group of LUs).

Path control implements the logical unit description (LUD) as the secondary LU. This view of the LUD as a port uncouples the end user application from a hard-coded physical address and allows the operating system the option of late coupling of application and physical address. Path control also treats all paths in a parallel fashion to provide flexibility in the host's use of the logical links.

For outbound messages, path control constructs a path information unit in the output buffer and sends an output request SRM to the secondary SDLC IOM.

For an input path information unit received from the secondary line IOM, path control ensures that the message unit is large enough to contain the transmission header and request/response header and that the transmission header is a format identification type 2. The origin and destination address fields in the transmission header are used to determine the appropriate session (SSCP-PU, SSCP-LU, LU-LU); if the PU is active, the path information unit is routed to the connection point manager.

Transmission Control


The transmission control element is a composite protocol machine that provides control for a locally supported half session. Transmission control consists of the following SNA components.

Common Session Control: The common session control provides common support for handling flows to half sessions that are not active. Common session control directs an appropriate activation request to session control for further processing.

Connection Point Manager: The connection point manager is the control point within the LU for distribution of request/response units, validating input sequence numbers, maintaining the pacing state machines, and supporting other functions related to the half-session flows.

For outbound messages, the session and data traffic states are checked. Additional checking is performed before the message is sent to path control:


- On the expedited flow, the connection point manager forwards a request only if no response to a previously expedited request is due from the host.
- On the normal flow, the size of the request/response unit is validated.
- When secondary-to-primary pacing is supported for normal flow request, the connection point manager determines when to set the pacing indicator in the request header to indicate that a pacing response must be returned by the host. The connection point manager then prevents the forwarding of additional outbound normal requests until the pacing response is received.
- When primary-to-secondary pacing is supported, and if a pacing request has been received, the connection point manager sets the pacing indicator on in the response header in a normal flow response to notify the host that additional normal flow requests may be sent.



When session and data traffic states are not active, the connection point manager forwards the permitted inbound SNA activation requests (bind and start data traffic) to common session control. For other inbound messages, the connection point manager checks the request/response header to determine if paced or expedited requests can be sent. When a pacing response should be sent to the host, the size of the request/response header is validated, the sequence number in the transmission header is compared with the expected value, and a flag is set to indicate that a pacing response should be sent.

Session Control: Session control supports protocols related to session and data traffic activation, de-activation, and recovery. All session control requests flow from the primary unit to the secondary System/38, and all responses flow from the secondary System/38 to the primary unit.

For an outbound response, session control identifies the response code and sets the appropriate state indicators according to whether the response is positive or negative. The response is forwarded to the path control.



The secondary IOMs support the following SNA session control requests:

- Activate PU to establish an SSCP-PU session.
- De-activate PU to break an SSCP-PU session.
- Activate LU to establish an SSCP-LU session.
- De-activate LU to break an SSCP-LU session.
- Bind to establish an LU-LU session.
- Unbind to break a LU-LU session.
- Start data traffic to permit user data to flow in the LU-LU session.
- Clear to stop user data flowing in the LU-LU session.

Requests pertaining to LU-LU sessions are sent to the secondary end user via I/O requests for additional processing.

I/O Support

Output

The output process is started by a request I/O queuing function (#T2RQIO). This function receives all I/O requests from the secondary station IOM queue and enqueues the requests on the correct LU queue. The requests are always enqueued last on this queue and are categorized according to the I/O path on which they are to be executed (for example, expedited or normal transmit, expedited or normal receive, receive-any). The queuing function uses the second byte of the message key for these encodings.

Once the requests are enqueued, the scheduler (#T2SCED) is invoked to process the output. The scheduler consists of two parts: loop selection and logical I/O path selection. Loop selection uses the half-session control block as its basic unit. This module multiplexes output for different LUs into one output request and ensures that each LU has equal opportunity for output.

The logical I/O path selection routine (#T2BSTO) gains control from the loop selection module when a half-session control block has I/O requests to be processed. This selection locates the next request to be processed (for example logical I/O path), enforcing the path control rule that expedited processing is executed before normal processing. The selection routine also controls the building of the SNA frames.

The transmit path message pointer in the half-session control block is examined, and if the pointer is 0, the appropriate 2-byte key is built and the LU queue is searched to find an I/O request. If an I/O request is not located, the process returns to the loop scheduler, and another LU request is processed. If an I/O request is located, the output SNA frames and associated buffer control lists are built, and control is returned to the loop scheduler.

On return, the loop scheduler determines if it is time to build an output request. If not, the loop scheduler selects the next half-session control block and repeats the selection sequence. If the output request is to be built, the loop scheduler builds the request, puts the buffer in a busy status, and sends the output request SRM to the secondary SDLC IOM.

Output Posting

The secondary station IOM now waits for the output to complete or for the input to arrive; output requests have a higher priority than input requests. A transmit/receive Request I/O instruction is first encoded as a transmit Request I/O instruction; then, when all transmits are complete, the instruction is encoded as a receive-only Request I/O instruction and enqueued first on the LU queue. The output request response process uses the half-session control block pointer to locate the Request I/O instruction associated with the frame. Then, using the indexes in that entry, it locates the request descriptor (RD) associated with the SNA frame and marks it processed.

At this point the Request I/O instruction can be in three states:

- More transmit RDs to process
- All transmits complete, but receives yet to process
- All transmits and receives complete

The appropriate action is then taken by the secondary station IOM to complete the output request response process. At this time, the buffer is marked not busy.

On return, the loop schedule is invoked to schedule more activity.

Input

The secondary station IOM does not explicitly request input from the transmission line. The secondary SDLC IOM responds to polls from the primary unit and, in effect, the secondary station IOM has a read operation outstanding. The secondary SDLC IOM passes received information frames to the secondary station IOM via an input message. This message contains the location and number of valid information frames stored in the input buffer. The input routine processes the information frames one frame at a time until all frames have been processed. The input routine uses the transmission header to locate the correct half-session control block and then determines the logical I/O path on which the frame is to be sent. If an I/O request is not pending or if no buffer space is available, the frame is considered unsolicited data, and an event or a feedback record is returned to the user. The user is informed of only the first frame of unsolicited data; however, the subsequent frames can cause the same situation, depending on whether the I/O request contained enough buffer space to contain the frames.

Unsolicited data is held on the queue of the conversation control block in the form of a message. This message is built by the secondary station IOM and has the same format as an input message. The message also contains a data area for the frame, allowing the secondary station IOM to free the input buffer of the secondary line IOM even if a user buffer for the data does not exist. SDLC unsolicited data is processed as an input request by the same modules that perform input processing. The conversation control block contains indicators to inform the scheduler to process the data.

Error Logging

The secondary station IOM does not keep statistics concerning the station; these statistics are kept by the secondary SDLC IOM. The secondary station IOM does log SNA path errors. These errors are recorded using the format of the error recording functions. The data contains portions of the SNA frame, including the transmission and request headers, the sense data, and the first 14 bytes of the request/response unit.

DATA AREAS

Refer to *Source/Sink Data Areas* in the *Vertical Microcode Overview* section of this manual for descriptions of source/sink data areas. Also, refer to *Data Areas* in the *Instruction Processors* section of this manual for descriptions of the ND, CD, and LUD.

Network Architecture Control Block

The network architecture control block is a common control area used to manage the systems network architecture (SNA) portions of the secondary station. It contains a subset of data in the CD and is always in main storage when the secondary station task is executing. The network architecture control block functions as a directory to areas built in machine-wide storage. It also functions as a collection point for vital converged station characteristics, a control point for converged station output, and a common location for unique SNA and converged station work areas.

The network architecture control block is allocated from machine-wide storage at vary-on CD time.

Logical Unit Name Table

The logical unit name table is used as a collector for all information necessary to operate the logical unit. This entry represents the LU such that the local LU and the remote LU can communicate through logical groupings of conversations.

The logical unit name table is accessed externally through the system pointer to the LUD and accessed internally through the LUD I/O index contained in the LUD. The LUD I/O index is set in the LUD and in the conversation identifier at vary-on LUD time. The LUD I/O index is a direct index into the logical unit name table. The first logical unit name table entry represents the physical unit. It is accessed with an index of 0.

Each logical unit name table entry contains the LU type 1 for purposes of routing to the proper SNA modules and building the conversation identifier for LU1.

The logical unit name table serves as an anchor for groupings of SNA sessions and the conversations used to access the SNA sessions. All half-sessions have a preassigned local address which is the destination address field in the LUD. The storage for the mode table and conversations is allocated in machine-wide storage. This is done at vary-on CD time for LU1.

Each logical unit name table entry representing a LUD has one mode entry. Each mode entry has two conversations representing the SSCP-LU session and the LU-LU session. The SSCP-LU conversation has a conversation index of 0.

Mode Table

The mode table is used for accessing the SSCP-LU and LU-LU conversation.

Half-Session Control Block

Half-session control blocks are used for routing transmit and receive requests to the proper network addressable unit and for routing input to the proper destination in the System/38. Each half-session control block represents an SNA session; supports the SNA transmission subsystem; and contains fields that support path control (routing and expedited/normal SNA paths), connection point manager (pacing), and LU (I/O queueing and session states).

Half-session control blocks are allocated and initialized in machine-wide storage at CD vary-on time for LU1 LUDs.

Each half-session control block represents an SNA path for data transmission.

The first two half-session control blocks are reserved for boundary function support and internal secondary station IOM usage. The first supports an SSCP-physical unit session that is activated and ready for I/O traffic at CD vary-on time. The second half-session control block is used as a temporary output holding area when the output structure is already busy processing an output request.

Conversation Control Block

The conversation control block represents the conversation resource. The LU1 LUDs have two conversation control blocks associated with the two LU1 SNA sessions (SSCP-LU and LU-LU).

The conversation identifier is used to access the conversation for all machine-interface operations, and the backward pointer in the half-session control block is used to access the conversation for all input operations.

STRUCTURE

The following is a list of the secondary station and SDLC secondary IOM modules and the function that each module performs. The list also shows how the module is invoked. The VMC components that can be invoked within each module are also shown.

#NA2CPLU Activate/De-activate LU Response Processor

Function: Updates LU state machines for activate and de-activate LU requests.

How Invoked: Within this component.

#NA2CPMR Connection Point Manager Receiver

Function: Performs connection point management checks (including pacing) for data received and routes for further processing.

How Invoked: Within this component.

#NA2CPMS Connection Point Manager Sender

Function: Performs connection point management checks for data to be transmitted.

How Invoked: Within this component.

#NA2CPPU Activate/De-activate PU Response Processor

Function: Updates PU state machines for activate PU and de-activate PU requests.

How Invoked: Within this component.

#NA2EROR SNA Error Processor

Function: Determines if negative response must be returned, and if so, builds negative response unit and routes for transmission.

How Invoked: Within this component.

#NA2PCR Path Control Receiver

Function: Validates transmission header data and routes path information unit received to connection point manager receive.

How Invoked: Within this component.

#NA2PCSD Path Control Sender

Function: Builds path information unit in output buffer and denotes output is pending.

How Invoked: Within this component.

#NA2RFMS Request/Record Formatted Maintenance Statistics

Function: Responds to REQMS request received, if necessary, then builds record formatted maintenance statistics request as a pseudo-I/O request for transmission.

How Invoked: Within this component.

#NA2SCSD Session Control Sender

Function: Identifies session control response, updates appropriate finite state machine states, and forwards response to path control send.

How Invoked: Within this component.

#TP2SECS Secondary Station IOM Activity Controller

Function: Loops to receive an SRM from the secondary station IOM send/receive queue and routes it for appropriate processing.

How Invoked: Other VMC components.

#T2ALUR Activate LU Response Processor

Function: Interrogates activate LU-received response from MSCP to establish SSCP-LU and LU-LU half-session controls and initializes transmission and response headers for SNA activate LU response.

How Invoked: Within this component.

#T2APUR Activate PU Response Processor

Function: Interrogates activate PU-received response from MSCP to establish output structure storage and initializes transmission and response headers for SNA activate PU response.

How Invoked: Within this component.

#T2BDRI Build Dummy I/O Request

Function: Creates a pseudo-I/O request structure and enqueues it to the dummy half-session control block for transmission.

How Invoked: Within this component.

#T2CMDS Route Secondary Station Commands

Function: Interrogates all non-I/O SRMs and response SRMs and routes each to the appropriate routine. Contains the following internal routines:

- **#T2ACTS:** Processes activate session SRM.
- **#T2DACS:** Processes de-activate session SRM.
- **#T2VOFL:** Processes vary-off LUD SRM.
- **#T2VOFC:** Processes vary-off CD SRM.
- **#T2VONL:** Processes vary-on LUD SRM.
- **#T2CONT:** Processes request I/O (continue) SRM.
- **#T2QUSC:** Processes quiesce SRM.
- **#T2RSET:** Processes reset SRM.
- **#T2RSUM:** Processes resume SRM.
- **#T2SRTO:** Processes send/receive time-out SRM.
- **#T2SPND:** Processes suspend SRM.

How Invoked: Within this component.

#T2DIS Process SNA Cleanup for SDLC
Disconnect Command Received

Function: Sends abnormal disconnect SRM to MSCP for normal and abnormal disconnect SRMs. Sends abnormal de-activate PU SRM to MSCP for the second Set Normal Response Mode SRM.

How Invoked: Within this component.

#T2DOWN Report Station Failure Condition

Function: Marks a station failure condition and reports it via a feedback record if a Request I/O instruction is outstanding.

How Invoked: Within this component.

#T2ERLG Build Error Log Entry

Function: Builds an error log entry SRM and sends it to be logged.

How Invoked: Within this component.

#T2FDBK Feedback I/O Request Result

Function: Builds the feedback record for an I/O request feedback SRM.

How Invoked: Within this component.

#T2HFDBK DHCf I/O Request Feedback Processor

Function: Feeds back an I/O request built by the DHCf code in the secondary station IOM.

How Invoked: Within this component.

#T2HIN 525x Emulation Input Processor

Function: Converts inbound 327x data streams to a 525x data stream.

How Invoked: Within this component.

#T2HIPIU DHCf Input Path Information Unit Processor

Function: Processes input data from a System/370 terminal for DHCf.

How Invoked: Within this component.

#T2HOUT 525x Emulation Overview for Output

Function: Converts outbound 525x data stream to 327x data stream.

How Invoked: Within this component.

#T2HOUTR DHCf Output Request Response Processor

Function: Completes an output request for a DHCf transmit I/O Request

How Invoked: Within this component.

#T2HPOFF DHCf LUD Power-off Simulator

Function: Terminates the DHCf half session.

How Invoked: Within this component.

#T2HRQIO DHCf I/O Request Processor

Function: Processes all I/O requests from the work station function manager for the DHCf devices.

How Invoked: Within this component.

#T2IOS Process Inoperative State Request Received from Secondary Line IOM

Function: Sets station off line and calls #T2DOWN to notify the machine interface of a station failure.

How Invoked: Within this component.

#T2IPIU Input Path Information Unit Processor

Function: Routes received SNA path information unit for validation, then moves data into receiving I/O request buffer area.

How Invoked: Within this component.

#T2OUTR Output Request Response Processor

Function: Matches output request response to I/O Request and sends feedback record when processing is complete.

How Invoked: Within this component.

#T2OUTX Forward Output Request to Secondary Line IOM

Function: Completes and sends output request SRM to the secondary line IOM.

How Invoked: Within this component.

#T2REQB Service Routine to Build a DHCP I/O Request Message

Function: Builds an I/O request message for DHCP to convert the work station function managers I/O request into an LU1 I/O request for transmission of data.

How Invoked: Within this component.

#T2RQIO I/O Request Processor

Function: Enqueues an I/O request with the correct flow key to the proper half-session control block queue.

How Invoked: Within this component.

#T2SCED Activity Scheduler

Function: Scans half-session control blocks to locate and dispatch transmit I/O requests.

How Invoked: Within this component.

#T2SEH Secondary Station IOM Exception Handler

Function: Identifies the exceptions encountered in the secondary station and returns them to the mainline RECM in #TP2SECS when the exception is caused by the user.

How Invoked: As fourth level exception handler.

Primary Station I/O Manager

INTRODUCTION

Primary station I/O manager (IOM) is a VMC task that interfaces with:

- Error log
- SDLC or X.25 line IOM
- Machine services control point (MSCP)
- Modify Controller Description instruction processor
- Modify Logical Unit Description instruction processor
- Request I/O instruction processor

The user of the primary station IOM can execute Modify Controller Description and Logical Unit Description instructions, and make, break, and manage a systems network architecture (SNA) (T3) path to a logical unit (LU) attached to the station. The Request I/O instruction is used to communicate with that LU. The primary station IOM is the multiplexing point for all communications to and from the station.

There is one primary station IOM task per station on the line.

The interface to the primary station IOM is via send/receive messages and the feedback record. Refer to the *Vertical Microcode Overview* section of this manual for a description of these areas.

Figure 31-1 shows the operation of the primary station IOM. The router module receives control when the primary station IOM task is created at vary-on controller description (CD) time. The router issues a receive to the primary station IOM queue and waits for a message to be enqueued. When a message is enqueued, the router invokes the module that performs the requested function. If the requested function is not valid, the primary station IOM returns the message with an unsupported status indicated. If the message is a response requiring no further processing, the primary station IOM frees the storage occupied by the message (this message was generated by the primary station IOM), issues a receive to the primary station IOM queue, and waits for the next message to be enqueued.

There is a separate primary station IOM module for each unique function code and response. The modules return a response only if further processing is required.

There are three types of primary station IOM modules:

- System/38 instructions
- SNA
- I/O

The System/38 instruction support modules recognize the requested function and initiate the processing of that function. These modules are further categorized into I/O, path, and MSCP support functions.

The SNA support modules are part of the input and output data path. The SNA support is used with the Request I/O instruction, since primary station IOM provides the interface that is compatible with SNA/SDLC-type devices.

The I/O support modules provide support of the Request I/O instruction (this support includes the SNA modules).

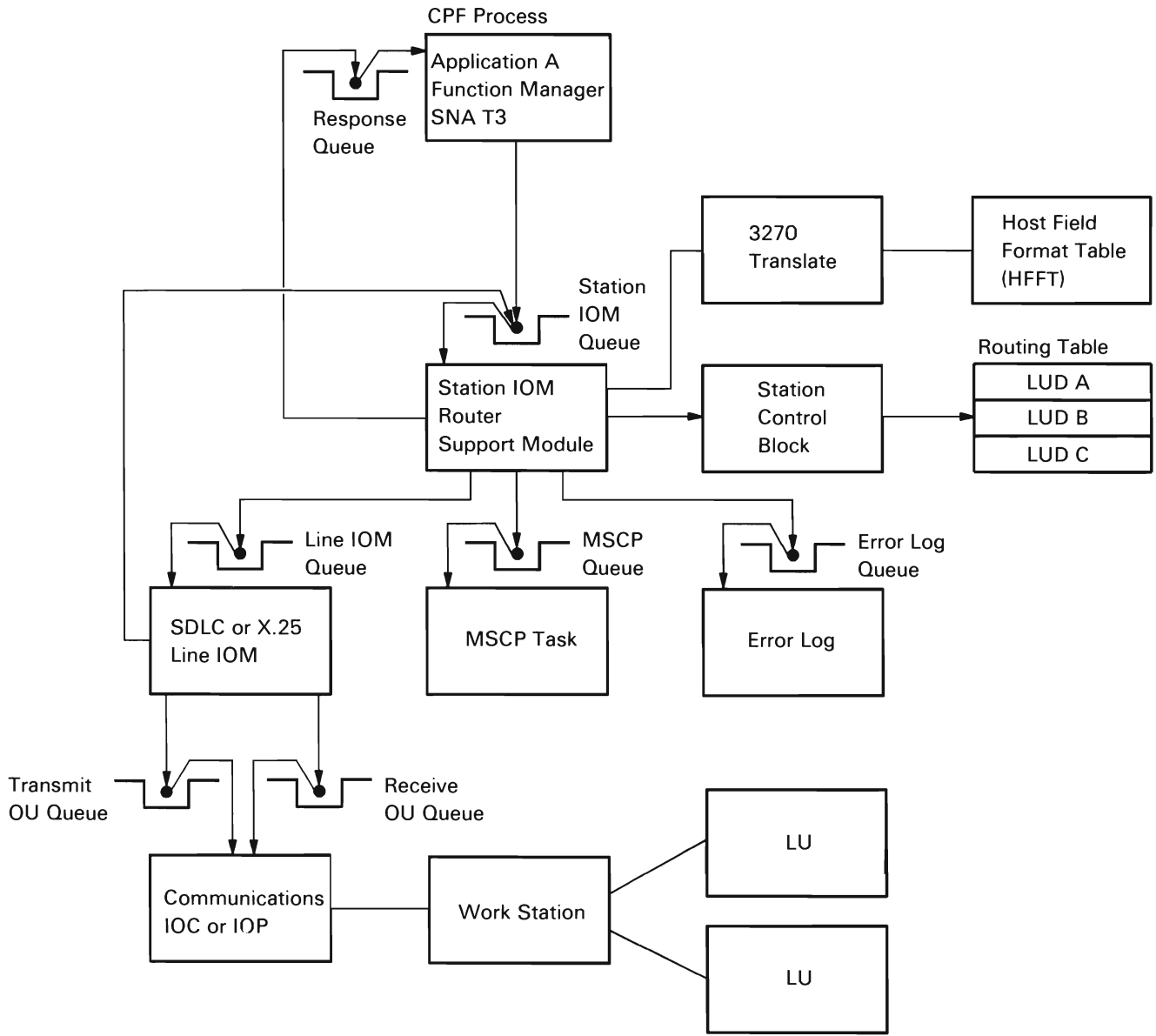


Figure 31-1. Primary Station IOM Processing

System/38 Instruction Support

All support of the System/38 instructions is presented to station IOM in the form of a VMC message. These messages are the result of mapping the instructions to the VMC format. This step is performed by the high-level language interface, instruction layer.

All messages are routed to their support modules by the primary station IOM router. The instructions supported are described in the following paragraphs.

Modify Controller Description (Vary-On/Off): This instruction is used to establish or break the communications path to the physical unit in the station represented by a CD.

Modify Controller Description Error Recovery Mode (Vary-On/Off): This instruction is used to place the communications path to the physical unit in error recovery mode represented by the CD.

Modify Logical Unit Description (Vary-On/Off): This instruction is used to establish or break the communications path from the MSCP to the LU. The instruction also performs initial setup for the LU-to-LU path.

Modify Logical Unit Description Error Recovery Mode (Vary-On/Off): This instruction is used to place the communications path from the MSCP to the logical unit in error recovery mode represented by the LUD.

Modify Logical Unit Description (Activate/De-activate): This instruction is used to establish or break the communications path from the system LU to the station LU.

Resume: A Modify Logical Unit Description (activate) instruction is altered to a resume command by the instruction layer when the LU is in the inactive state (quiesce, suspend, reset).

Modify Logical Unit Description (Quiesce): Once the primary station IOM receives this request, it completes all I/O requests that are either in process or waiting on the queue. When this instruction is completed, device management reaches a normal completion and all I/O request processing is stopped until a Modify Logical Unit Description (activate) instruction is issued.

Modify Logical Unit Description (Reset): This instruction causes all I/O requests to be returned to the user return queue, with an indicator that the request was not processed. The number of request descriptors (RDs) processed is also set in the feedback record. This instruction also destroys all available unsolicited data and places the communications path in an inactive state. The path can be reactivated with a Modify Logical Unit Description (activate) instruction.

Modify Logical Unit Description (Suspend): This instruction ensures that all I/O requests for logical units are in the suspend state. This means that all the I/O requests that have been started will be processed to a transmit/receive RD boundary. The I/O requests that complete are put on the return queue with the appropriate feedback code and the number of RDs processed. Any transmit/receive type of I/O request with only the transmit portion complete remain on the logical unit queue.

Request I/O: This instruction is used for two purposes:

- Perform I/O on the various paths such as MSCP-Physical Unit (PU), MSCP-LU, LU-LU.
- Return the logical unit to an active path state after a terminating error has occurred (Request I/O-Continue instruction).

The I/O function is described under *I/O Support* in this chapter. The Request I/O (continue) instruction uses a terminating error mode bit in the routing table. This bit is set when a terminating error is encountered. The user is informed of this action by a code in the feedback record. If the user can recover from the error, the user resets the mode bit with the Request I/O (continue) instruction, and normal processing resumes.

SNA Support

The primary station IOM acts as the transmission subsystem layer of SNA 0081 (T3). The SNA support consists of path control, connection point manager, and I/O for the MSCP.

Path Control

Path control uses the information in the T3 transmission header to control delivery of received basic information units to either the specified supervisory services or a LU in a node (including a clustered group of LUs).

Path control is broken into the following areas:

- Routing
- Format identification translation
- Logical I/O paths

The routing function consists of a search of the routing table. In the outbound direction, a routing table index and the request I/O function (MSCP or normal) are used to locate the correct path (MSCP-PU, MSCP-LU, LU-LU). The index is set up by the primary station IOM at vary-on LUD time and placed in the instruction by the instruction layer.

In the inbound direction, the destination address from the transmission header is used to locate the correct logical unit in the routing table. The transmission header flow bits are then used to point to the LU-LU or MSCP-LU path. For either inbound or outbound traffic, once the correct routing table entry is located, the primary station IOM performs the necessary steps to deliver the basic information unit.

Format identification translation is the process whereby the SNA transmission header is converted to the System/38 internal format and the System/38 internal format is converted to the SNA transmission header. This function also provides all of the SNA 0081 validity checks on the T3 transmission header. The translation modules also initiate all logging and negative response actions.

Once the translation and routing functions have located the correct path (for example, routing table entry), primary station IOM selects the correct logical I/O path for the frame. The frame or I/O request contains the data necessary to identify the correct logical path. There is a based structure that maps the fields describing an I/O path; the path is selected by setting the basing variable. SNA defines the logical I/O paths as either expedited or normal for both transmission and receive paths. These logical I/O paths are represented in each routing table entry.

Connection Point Manager

The connection point manager functions are divided between the VMC and the control program facility (CPF). The primary station IOM in VMC provides the pacing function. This function provides support equivalent to that of SNA 0081. Pacing is a means whereby the receiving connection point manager can control the rate at which it receives requests on the normal data flow. Responses and expedited data are not paced, only outbound data is paced.

Each routing table entry has fields used for pacing. These fields are the pacing count, count of frames sent, and a pacing flag.

Pacing is started by setting the pacing count (limit) to the value contained in either the logical unit description (LUD) or a negotiable-bind response from a terminal, and by setting on the pacing flag. The primary station IOM then begins sending path information units and incrementing the counter until the counter equals the pacing limit (this occurs even if the pacing flag has been set on by a response from a terminal). The pacing response from the terminal sets the pacing flag indicating that the terminal can handle N more path information units (N being the pacing limit). The terminal controls the flow via sending the pacing response when it is ready to receive more frames.

MSCP Functions

The primary station IOM provides the MSCP with an interface similar to the one provided to the logical unit. Following are the differences that apply to this interface:

- The MSCP-LU or MSCP-PU path is not placed in terminating-error mode. Thus, a continue is not required to resume request I/O operations.
- Feedback records are in the format of posted messages.
- If a quiesce, suspend, or reset command is used, the primary station IOM assumes that no additional I/O request will be issued until a resume request is issued.

Unsolicited data is reported to the MSCP through VMC messages. The MSCP either processes the data or passes it to the user interface as event-related data. The events used are formatted or unformatted supervisory services data. The MSCP always returns the unsolicited data message with a successful completion code.

I/O Support

Output

The output process is started by a request I/O queueing function (#TPSRQIO). This function receives all I/O requests from the primary station IOM queue and enqueues the requests on the correct LU queue. The requests are always enqueued last on this queue and are categorized according to the I/O path on which they are to be executed (for example, expedited or normal transmit, expedited or normal receive, receive-any). The queueing function uses the second byte of the message key for these encodings.

Once the requests are enqueued, the scheduler (#TPSCHED) is invoked to process the output. The scheduler consists of two parts: loop selection and logical I/O path selection. Loop selection uses as its basic unit one routing table entry. This module multiplexes output for different LUs into one output request and ensures that each LU has equal opportunity for output.

The logical I/O path selection module (#TPSBSO) gains control from the loop selection module when the next routing table entry that has request I/Os to be processed is selected. This selection locates the next request to be processed (logical I/O path, for example), enforcing the path control rule that expedited processing is executed before normal processing. The selection routine also controls the building of the SNA frames.

The correct path is selected from the transmit path message pointers in the routing table entry. If the pointer is 0, the appropriate 2-byte key is built and a dequeue-equal is done on the LU queue. If a request I/O is not located, the process returns to the loop scheduler and another LU request is processed. If an I/O request is located, the output SNA Frames and associated buffer control lists are built and control is returned to the loop scheduler.

For 3270 and System/38 finance type devices, module #TSTOUT is called to translate the 5250 data stream to 3270 or System/38 finance data stream before the SNA frames are built.

On return, the loop scheduler determines if it is time to build an output request. If not, the loop scheduler selects the next entry and repeats the selection sequence. If the output request is to be built, the loop scheduler builds the request and puts the buffer in a busy status.

Output Posting

The primary station IOM now waits for the output to complete or for input to come in; output requests have a higher priority than input requests. A transmit/receive Request I/O instruction is first encoded as a transmit; then, when all transmits are complete, the instruction is encoded as a receive-only and enqueued first on the LU queue. The output request response process uses the buffer control list routing table pointer to locate the Request I/O instruction associated with the frame. Then, using the indexes in that entry, it locates the RD associated with that SNA frame and marks it processed.

For Kanji devices, module #TPSIDO (for display output) or module #TPSIPO (for printer output) is called to translate ideographic extension characters into RAM addresses and to load the ideographic RAMs into the device.

At this point, the Request I/O instruction can be in three states:

- More transmit RDs to process
- All transmits complete, but receives yet to process
- All transmits and receives complete

The appropriate action is taken, thus completing the output request response process. At this time, the buffer is marked not busy.

On return, the loop scheduler is invoked to schedule more input.

Input

The primary station IOM does not explicitly request input from the line. Line IOM polls the stations for input and, in effect, the primary station IOM has a read outstanding. Line IOM passes the result of the polling to the primary station IOM via an input message. This message contains the location and number of valid information frames stored in the input buffer. The input routine processes the information frames one frame at a time until all frames are processed. For Kanji displays, module #TPSID1 is called to translate RAM addresses in the data stream back into ideographic extension characters, and to handle ideographic alternative entry LUSTAT. The input routine uses the transmission header to locate the correct routing table entry; then it determines the logical I/O path on which the frame is to be sent. For 3270 and System/38 finance type devices, module #TSTIN is called to translate the 3270 data stream to 5250 data stream before the data is given to CPF. If an I/O request is not pending or if no buffer space is available, the frame is considered unsolicited data and an event or feedback record, respectively, is returned to the user. The user is informed of only the first frame of unsolicited data; however, subsequent frames can cause the same situation, depending on whether or not the I/O request contained enough buffer space to contain the frames.

Unsolicited data is held on the queue of the logical unit in the form of a message. This message is built by the primary station IOM and has the same format as an input message. The message also contains a data area for the frame, allowing the primary station IOM to free the input buffer of line IOM even if a user buffer for the data does not exist. Unsolicited data is processed as an input request by the same modules that perform input processing. The routing table entry contains indicators to inform the scheduler to process the data.

Error Logging

The primary station IOM does not keep statistics concerning the station; these statistics are kept by line IOM. An exception is for Kanji controllers, in which case some Kanji statistics are kept by the station IOM. SDLC station IOM does log SNA path errors. These errors are recorded using the format of the error recording functions. The data contains portions of the SNA frame, including the transmission and request headers and the first 14 bytes of the request/response unit.

DATA AREAS

The primary station IOM has two major data areas: the station control block and the routing table. It also has ownership of the output buffers of the station. The input buffers are owned by line IOM and are processed using an input message. For 3270 devices, an additional area called the host field format table (HFFT) is built and maintained by the PSIOM.

Station Control Block

Figure 31-2 shows an overview of the station control block. The station control block is built in the invocation work area (IWA) at vary-on CD time. This block is a common control point for managing the station. It contains a subset of the data in the CD and is always in main storage when the primary station IOM task is executing. The station control block also functions as a directory to areas built in machine-wide storage. It also functions as a collection point for vital station characteristics, control point for station output, and common locations for unique SNA and station work areas.

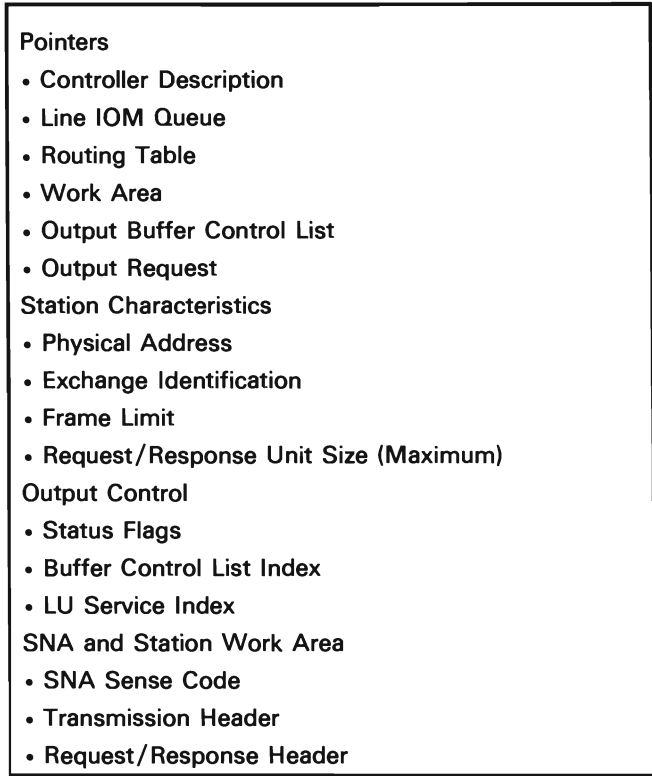


Figure 31-2. Station Control Block

Routing Table

The routing table is allocated and initialized in machine-wide storage at vary-on CD time. This table supports the SNA transmission subsystem and contains fields that support path control (routing and expedited/normal SNA paths), connection point manager (pacing), and logical unit (I/O queueing and session states).

An entry in the table represents an SNA path and a loop scheduling unit. Figure 31-3 shows an overview of a routing table entry. There are two entries for each logical unit attached to a station, one entry for the LU-LU path, and one entry for the MSCP-LU path. These entries are established at vary-on LUD time. The MSCP-LU path is activated and ready to process request I/O traffic. The LU-LU path is activated and ready to process request I/O traffic at modify LUD (activate) time.

Only the station has an MSCP path (MSCP-PU). This path is represented by a second routing table entry. This path is activated and ready to process request I/O traffic at vary-on CD time.

Routing table entries are established in pairs; there is only one spare entry for the first routing table entry number. This entry is activated and used for temporary LU-LU paths. (A temporary path exists for the time required to send an unrecognized destination address negative SNA response.)

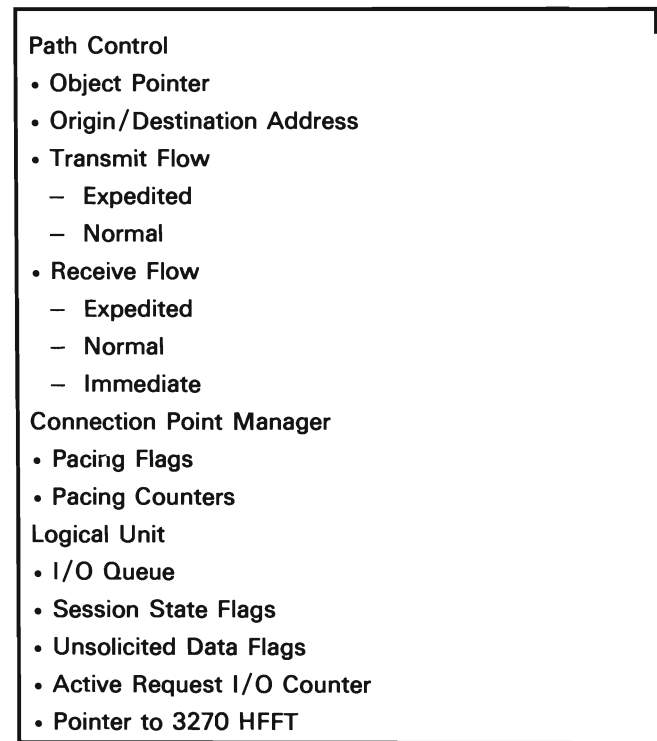


Figure 31-3. Routing Table Entry

3270 Host Field Format Table

This area is used for 327x and System/38 finance devices only. The area is created at either vary-on CD time for a controller or at vary-on LUD time for a device. The HFFT contains flags and pointers to control the translation and processing of data to and from 327x and System/38 finance controllers and devices. Figure 31-4 shows an overview of the HFFT table entries.

Frame Slot

The frame slot is a storage area used to build the SDLC/SNA frame for an output request. The slot is allocated on a doubleword boundary. The slot contains enough space to hold the address, control, longest transmission header, longest request/response header, and longest request/response unit. The slot is used by line IOM and enough slots exist to accommodate the maximum number of frames allowed for this station.

Buffer Control List

The buffer control list is allocated in machine-wide storage and consists of enough entries to accommodate the maximum number of frames allowed for this station. The entries in the list contain the data necessary to report the status of the output operation and build the output request that performs the output operation. Line IOM performs the functions necessary to build the operational program and handle the SDLC.

The next available slot in the list is located via the buffer control list index contained in the station control block. A slot is reset to zero when an output request is completed.

Host Field Format Table

- Device Type and Model
- Processing Flags
- Roll Data
- Pointer to Translate Storage
- Field Format Table Indexes
- Unsolicited Data Array
- Bind Save Area
- Outbound and Inbound Sequence Number Counters
- Field Format Table Index - Address and Attribute
- Printer Device Pointers and Counters

Figure 31-4. Host Field Format Table

Output Request Message

The output request message contains the output operation to be performed by line IOM. The information contained in this message consists of parameters that identify the station, number of frames, and the location of the buffer control list.

The primary station IOM ensures that only one output request is in process at one time by setting a buffer busy indicator in the station control block.

An output request can contain data for several logical units.

STRUCTURE

The following is a list of the modules in the primary station IOM and the function that each module performs. The list also shows how the module is invoked.

#TPSCHED LU Scheduler

Function: Selects routine table entries to be processed based on priority scheduling.

How Invoked: Within this component.

#TPSCMDS Group Primary Station IOM MSCP Commands and System/38 Instruction Processors

Function: Provides a means for grouping the primary station IOM MSCP commands and System/38 instruction processors into one module.

How Invoked: Within this component as a result of routing a function.

#TPSDOWN Station Down

Function: Reports to all LUs attached to this station that the station is down.

How Invoked: Within this component.

#TPSEXCP Station Exception Handler

Function: Recovers from an SSD destroyed exception and performs an orderly cleanup or any other exceptions.

How Invoked: From the third level exception handler.

#TPSFDBK Build Feedback Record

Function: Builds feedback record in order to report completion of a Request I/O instruction.

How Invoked: Within this component.

#TPSIGNL Signal Event

Function: Signals events for unsolicited data.

How Invoked: Within this component.

#TPSIPIU Input Path Information Unit Process

Function: Processes the input path information unit.

How Invoked: Within this component as a result of routing an input request.

#TPSPGLU Purge Logical Unit

Function: Purges unsolicited data for that logical unit and performs the modify LUD reset function.

How Invoked: Within this component.

#TPSRQIO Request I/O

Function: Enqueues an I/O request with the correct flow key on the proper routing table entry queue.

How Invoked: Within this component as a result of routing an I/O request.

#TPSIDO Ideographic Display Output

Function: Scans Kanji data, translates to RAM addresses, and manages ideographic RAM contents list.

How Invoked: Within this component.

#TPSIDI Ideographic Display Input

Function: Scans Kanji input data, translates RAM addresses into ideographic extension characters, processes alternative entry, LUSTAT.

How Invoked: Within this component.

#TPSIPO Ideographic Printer Output

Function: Scans Kanji data, translates to RAM addresses, and manages ideographic RAM contents list.

How Invoked: Within this component.

#TPSRCLM RAM Contents List Manager

Function: This set of subroutines performs common operations on the RAM contents list of ideographic devices.

How Invoked: Within this component.

#TPSROR Kanji Error Handler

Function: Handles errors unique to Kanji devices.

How Invoked: Within this component.

#TSTPRT 3270 Printer Translation

Function: Does outbound translation of 5250 printer data stream to 3270 printer data stream.

How Invoked: Called from #TSTOUT for printer device.

#TSTOUT 3270 Output Translation

Function: Does outbound translation of data from 5250 data stream to 3270 data stream.

How Invoked: Called from #TPSCHEM for 3270 devices only.

#TSTIN 3270 Input Translation

Function: Translates inbound data from 3270 data stream to 5250 data stream.

How Invoked: Called from #TPSIPIU for 3270 devices only.

#TPSSIOM Primary Station IOM Message Router

Function: This is the mainline loop for the primary station IOM. All incoming messages and responses are routed through this module.

How Invoked: Other VMC components.

#TPSTRNP Output Request Posting

Function: Posts the results of an output request and invokes the scheduler to continue processing.

How Invoked: Within this component as a result of routing an output request response.

System Control Adapter I/O Manager

INTRODUCTION

The system control adapter (SCA) I/O manager (IOM) provides an interface to the system control adapter. This interface is used by the machine services control point (MSCP), other IOMs, and S/38 instructions. High- and low-level messages are processed by the SCA IOM. A high-level message provides for the execution of an SCA function defined in the function address table (FAT). A low-level message provides for the execution of one or two operational programs supplied by the user.

An overview to SCA processing is shown in Figure 32-1. The SCA IOM is prebuilt in #RTTASKS and consists of an IOM task, an operational unit (OU) task, an IOM queue, an OU queue, and a queue control table (QCT). The SCA IOM receives the message and, for a high-level message, builds an operation request element (ORE) and the operational program based on the information in the function address table for the requested function. For a low-level message, an ORE is built for the user-supplied operational program. The ORE is sent to the OU queue and the SCA IOM executes a receive operation to the IOM queue to wait for the OU task to return the ORE. When the ORE is received by the SCA IOM and a second operational program is not supplied, the status field of the user request is updated and the user request message is returned to the user queue. If a second operational program is supplied, an ORE is built for that program and sent to the OU queue as before. The status is updated and the request message is returned to the user. The second operational program is executed regardless of the status of the first operational program. No retries are attempted for errors occurring during either operational program.

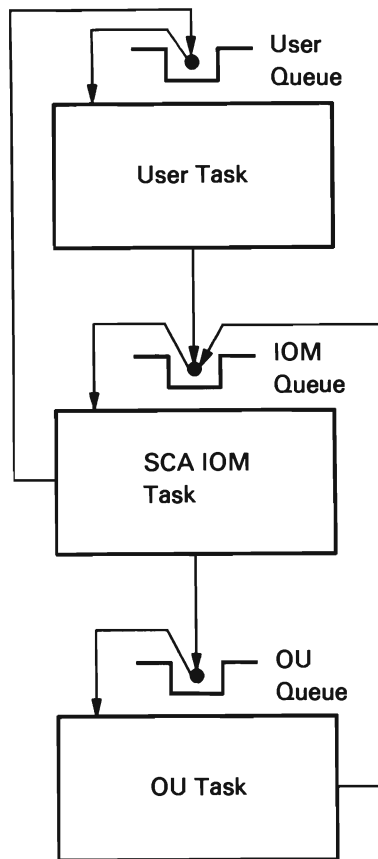


Figure 32-1. SCA Processing

Figure 32-2 shows the operation when the high-level functions requested involves a post event. The operation is similar to preceding operations with the following modifications:

- Receive the user message (which can include an address of a function-started message to be returned to the requestor's queue) when the post-event function is started.
- Build two operation programs and an ORE using information from the FAT and the user message. The last operation block of the first operational program is a message operation block to send the function-started message (optional) to the user queue.
- Send the ORE for the first operational program to the OU task to start the post-event function and to return the function-started message.

- When the ORE is returned to the SCA IOM, do a receive from the IOM queue to wait for the post-event message from the channel IOM.
- When the channel IOM post-event attention message is received, return the channel message and if needed send an ORE for the second operational program to the OU task to read the results from random access memory (RAM2) and to restore the SCA.
- Then the OU task returns the second ORE, the status field in the user message is set and the message is returned to the user queue.

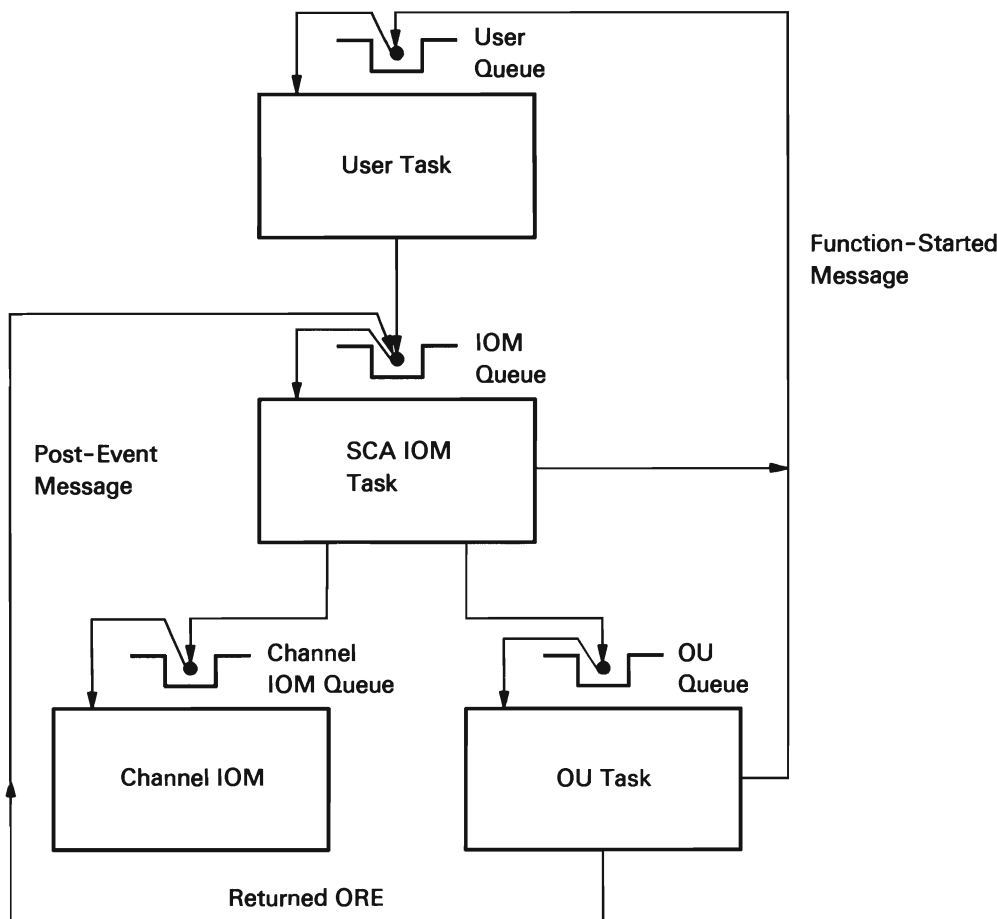




Figure 32-2. SCA Processing with Post Event



During the execution of a long running post-event attention function, the SCA IOM returns all incoming user requests (except for reset, terminate, and read sense) with a status field of busy.

A message is sent to the error log as the result of a basic status (BSTAT) error, post event error, channel hardware error, or an event handler error. If the post event error indicates a power fault, the power status is also logged.

The SCA IOM also processes device power fault indications. A device power fault indication is received by the SCA IOM as a channel post-event error message. The SCA IOM sends a message to the SCA IOM queue to obtain the power fault status and the status is logged.



When the system is connected to an uninterruptible power supply (UPS), the SCA IOM processes utility power failures and restorations. A utility power failure/restoration is received by the SCA IOM as a channel post-event error message when a full UPS is installed. The SCA IOM examines the machine communications area to determine if an event handler was defined for an uninterruptible power supply activation/de-activation event. If the event handler is defined, the SCA IOM signals a machine-wide uninterruptible power supply activated/de-activated event. If no event handler is defined, the SCA IOM saves volatile storage and powers down the system.

When a basic-UPS is installed, the SCA IOM will only get control if utility power returns. If it does get control (via a high priority message from the basic-UPS task), the SCA IOM calls the service task to update the time of day (TOD) clocks, checks for the event handler and if one is defined, signals a machine wide basic-UPS deactivated event. This event contains the amount of time spent on UPS.

The SCA IOM then issues a halt to the SCA OU and both console OUs, clears the console indicators (set by the SCA when interrogating the power line), reloads the resident RAM1 code, and sends a message to the error log that contains the amount of time spent on UPS.

DATA AREAS

User Message for SCA IOM

Communication with the SCA IOM is through the standard VMC send-receive message. The header portion of the message is defined by ZZSSVHDR, and appended to the header will be up to 22 bytes of additional information mapped by ZZLOSAMG. The format of the message is as follows:

VMC Header	Data Address	Data Length	Result Address	Result Length	Function-Started Message Address	Execute Data Address
------------	--------------	-------------	----------------	---------------	----------------------------------	----------------------

The format of the low level message is as follows:

VMC Header	First Operation Program Address	Second Operation Program Address (optional)
------------	---------------------------------	---

Function Address Table

The FAT is contained in #LOSAFAT and is used by the SCA to load and execute SCA routines. The FAT contains entries that describe functions being requested. The contents of the FAT are shown in Figure 32-3.

Resident Information <ul style="list-style-type: none">• Pointer to RAM1 Overlay• Number of RAM1 Transfers• Pointer to RAM2 Overlay• Number of RAM2 Transfers
First Function Entry <ul style="list-style-type: none">• Control Field• Reserved Area• Number of Channel Transfers• Pointer to Overlay• RAM1 Entry Pointer
Last Function Entry <ul style="list-style-type: none">• Control Field• Reserved Area• Number of Channel Transfers• Pointer to Overlay• RAM1 Entry Pointer

Figure 32-3. Function Address Table (FAT)

STRUCTURE

The following is a list of the modules in the SCA IOM and the function that each module performs. The list also shows how the module is invoked.

#LOSACNT SCA IOM Control

Function: Processes user requested SCA functions.

How Invoked: Other VMC components.

#LOSAFAT SCA Function Address Table

Function: Provides the information used to build operational programs for user messages. This module does not contain executable code.

How Invoked: Not applicable.

#LOSAIPL Initialize the SCA and Obtain Status

Function: Starts the SCA, reads the device status (DSTAT), restores RAM1 and RAM2, reads the position of the control switches, and reads the data storage that contains the soft errors encountered during an initial microprogram load (IMPL).

How Invoked: Initial program load (IPL) routine.



3270 BSC Emulation Management

3270 BSC emulation management is designed to communicate to a host system using 3270 type devices on a binary synchronous communications (BSC) line. The system/38 appears to the host as a 3271 controller with attached 3270 devices.

3270 BSC emulation management, shown in Figure 33-1, consists of two components, the BSC I/O manager (BIOM) for 3270 emulation and the 3270 emulation translation function.

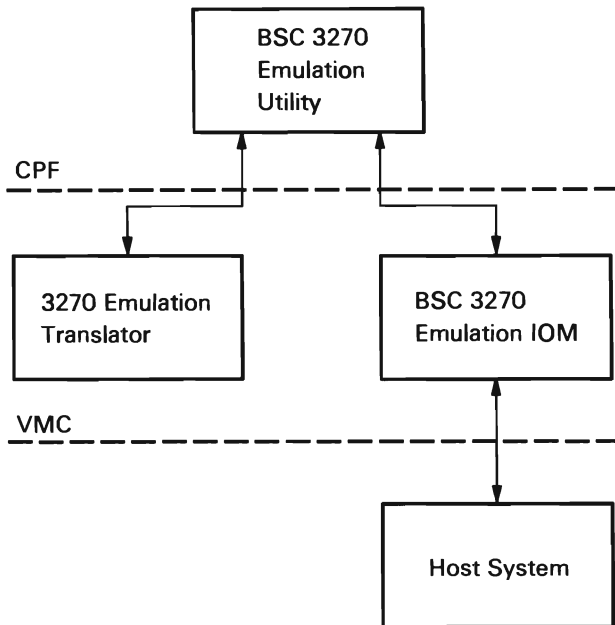


Figure 33-1. 3270 Emulation Manager Overview

The BSC I/O manager sends and receives 3270 data streams. The 3270 Emulation Translator translates 3270 data streams into 5250 data streams or program interface format. The 3270 device being emulated can be a display station or a printer.

BINARY SYNCHRONOUS COMMUNICATIONS I/O MANAGER FOR 3270 EMULATION

The binary synchronous communications (BSC) I/O manager (IOM) for 3270 emulation activates, manages, and de-activates the BSC telecommunications link and enforces BSC 3270 protocol. One BSC IOM for 3270 emulation task exists for each BSC 3270 emulation telecommunications link.

The 3270 BSC IOM interfaces with the following:

- The machine services control point (MSCP)
- The error log
- An I/O controller
- Diagnostic component
- Modify Network Description instruction
- Modify Controller Description instruction
- Modify Logical Unit Description instruction
- Request I/O instruction

A BSC IOM for 3270 emulation task is created by the MSCP as a result of a Modify Network Description (vary-on) instruction. The BSC IOM for 3270 emulation task is associated with one communications I/O controller (IOC) line position and is shown in Figure 33-2.

The BSC IOM for 3270 emulation is used to communicate with devices on a multipoint line as a tributary station. Up to 32 sessions per line can be supported.

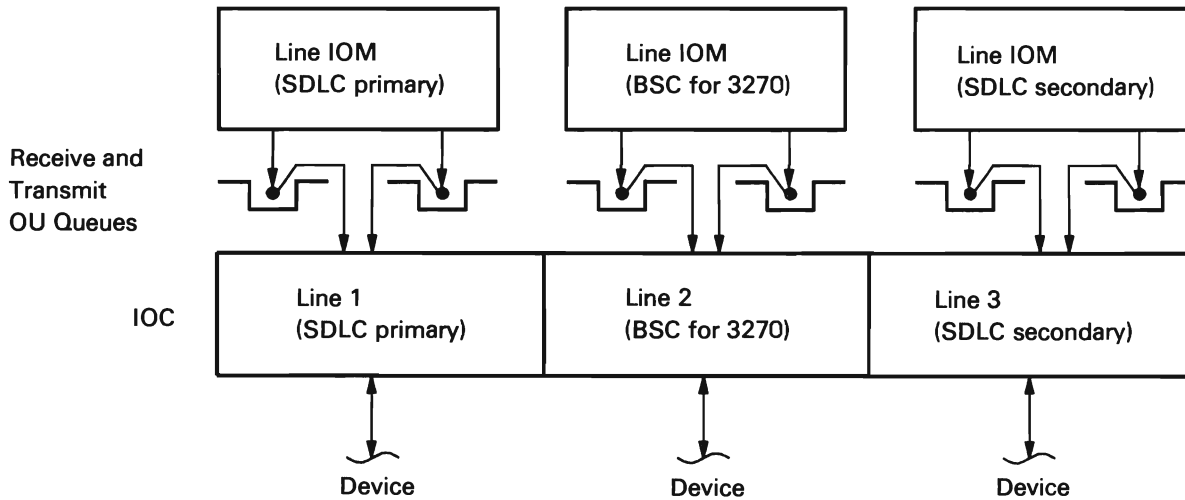


Figure 33-2. BSC IOM/IOC Line Position Relationships for 3270 Emulation

Communication with external components is through a send/receive message which the BSC IOM for 3270 emulation receives through a single send/receive queue as shown in Figure 33-3. The message can be generated in three ways; by an external VMC or diagnostic component, an operational request element (ORE), or a Request I/O instruction.

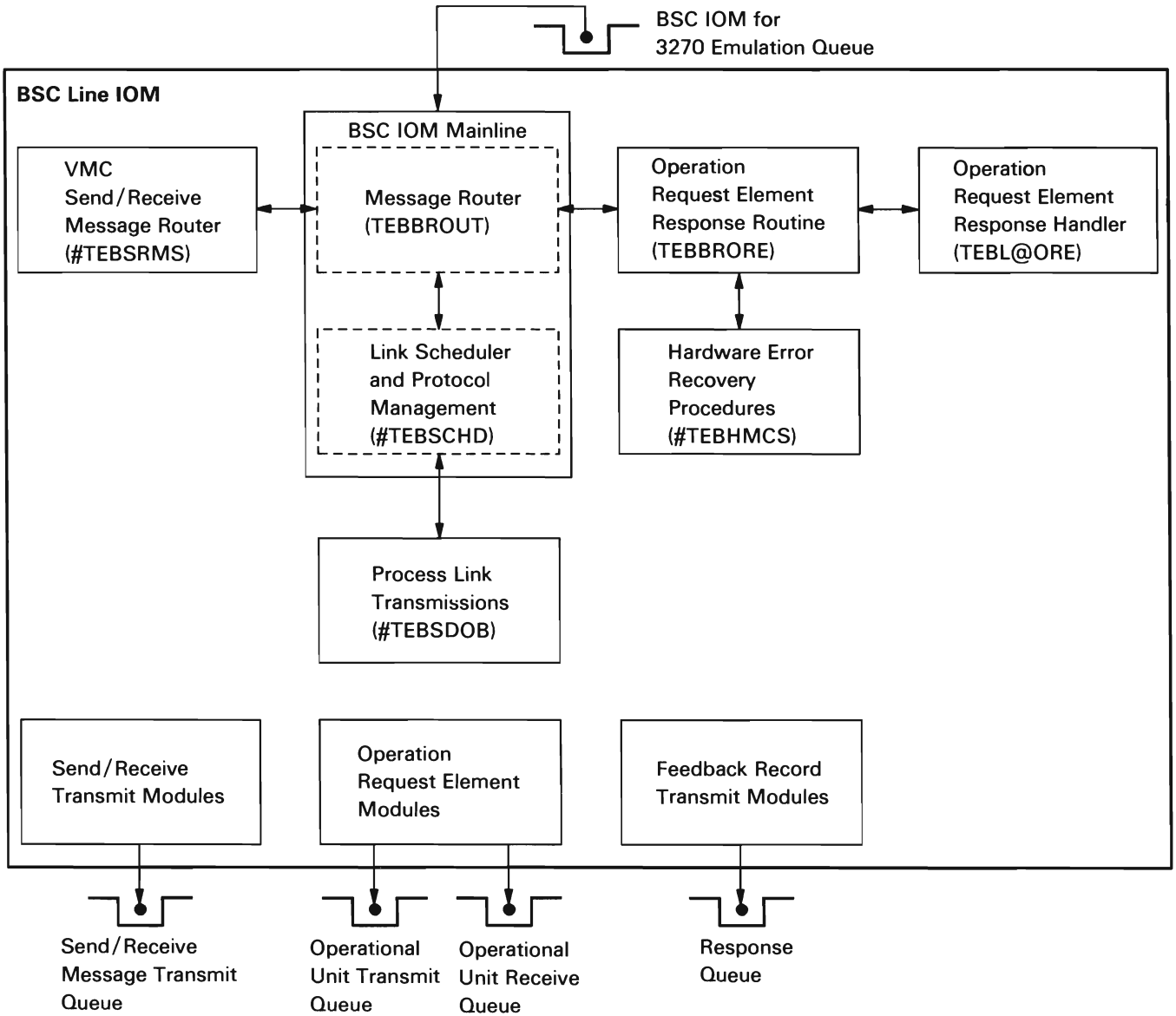


Figure 33-3. BSC IOM for 3270 Emulation Internal Structure

The message router receives the message and uses the key field in the message to determine if the message is generated from an external VMC component or diagnostic component, an ORE, or a Request I/O instruction. Then, based on the function field of the message, the message router invokes the appropriate message handler for the messages generated by the external VMC or diagnostic components. If the message is a Request I/O instruction then the message is queued on the internal session queue.

The messages generated by an external VMC component or diagnostic component, or Request I/O (continue) instruction messages are routed to a message specific routine (#TEBSRMS) for processing. See Figure 33-4 for a list of send/receive message (SRM) handling entry points.

Function	Routine (Entry Point)
Timer response	#TEBTIME
Activate session	#TEBASES
Request I/O (continue) instruction	#TEBRQIC
De-activate session	#TEBDSES
Reset session	#TEBRSES
Vary-off LUD	#TEBVOFL
Discontact	#TEBDCON
De-activate link	#TEBVOFN
Initialize line	#TEBENB
Vary-on LUD	#TEBVONL
Contact	#TEBCONT
Change network description retry sets	#TEBNDRT
Activate link	#TEBVONN
Exchange identification	#TEBXID
Machine interface timer message	#TEBSFTD
Modify device specific	#TEBMDSA
Modify unit specific	#TEBMUSC
Modify line specific	#TEBMLSC
Resume session	#TEBRSUM
Quiesce session	#TEBRSES
Suspend session	#TEBSPND
Read data store	#TEBRDS0
Internal trap	#TEBTRPO
Cancel Invite	#TEBSINV
Return message (invalid entry)	#TEBBADM

Figure 33-4. Send/Receive Message Handling Entry Points

The messages generated by the OREs are handled differently. The queue message router relies on the routine that generated the ORE to provide the address of the routine to process the ORE response. See Figure 33-5 for a list of ORE response handling entry points.

Function	Routine (Entry Point)
Perform 3270 BSC protocol analysis	#TEBTXTR
Perform 3270 BSC multipoint tributary analysis	#TEBOPMP
Write poll list response handler	#TEBWPL1
Perform 3270 response to status analysis	#TEBSTA1
Perform 3270 response to NAK sent after Read command received	#TEBTIM1
Process set line priority/reset initialize ORE response	#TEBVNN1
Initialize I/O controller ORE response	#TEBENB1
Perform disconnect response analysis	#TEBDCN1
Perform reset response analysis	#TEBRSL0
Read data store send/receive message handler	#TEBRDS0
Read data store ORE sender	#TEBRDS1
Read data store ORE response	#TEBRDS2
Trap send/receive message handler	#TEBTRP0
Trap ORE sender	#TEBTRP1
Trap ORE response handler	#TEBTRP2

Figure 33-5. ORE Response Handling Entry Points

The link scheduler and protocol management routines are invoked by the message router before the message is processed. The link scheduler and protocol management routines build the OREs for transmitting data or responses and sends the OREs to the operational unit queues.

All BSC 3270 IOM and IOC detected errors are processed by the error recovery routines.

3270 EMULATION TRANSLATION FUNCTION

Communication between a host system and the System/38 involves outbound (from the host) and inbound (to the host) data transmissions. The 3270 emulation translation function takes the outbound 3270 data stream and translates it to either a 5250 data stream or to the program interface, and takes the inbound data stream from either a 5250 or from the program interface and translates it to a 3270 data stream (see Figure 33-1).

The 3270 emulation translation function is invoked by the emulation utility through a Request I/O instruction which has an option to indicate a synchronous call/return request. The synchronous call/return request is a generalized function that calls a module based on information in the device LUD. The device-specific parameters in the LUD characterize the device to be emulated. These parameters are necessary to properly execute this Request I/O instruction. The source/sink request object also provides parameters in its SSR extension area for the translation process. These parameters include a pointer to the working space used by the translation function to maintain control information and the internal screen image buffer.

The translation process maintains a 1920-byte buffer (screen image buffer) for each session, which contains a representation of the 3270 screen. This buffer contains 3270 attributes and data at the same relative locations they would occupy in a 3270 display system. The buffer is updated as each inbound or outbound data stream is translated. Since 3270 attributes cannot be distinguished from data once they are separated from the original data stream, an array must be built in which each entry points to an attribute in the buffer. Together, the array and the buffer define the fields in the screen image buffer.

The translation process can result in either an outbound data stream for a display or printer, or in a format for a user program. The inbound translation *always* results in a 3270 data stream. For BSC, the program interface formats the outbound 3270 data stream into a screen image buffer, header and field information. If an error occurs during the translation process, an exception code is returned in the request I/O message.

DATA AREAS

Operation Request Element

The BSC IOM for 3270 emulation communicates with the IOC by way of a send/receive message called an ORE. In the operation block portion of the ORE the various commands are specified, data areas are indicated, and status is returned. Three types of operation blocks are used: the function operation block, the program operation block, and the message operation block.

The function operation block contains single commands such as initialize, line reset, write, and read commands.

The program operation block is used when multiple function operation blocks are to be executed. The program operation block references a chain of function operation blocks, each of which contains a command to be executed.

The message operation block is used during data transfer to eliminate the chance of command time-outs when two separate commands must be issued to the IOC for the execution of one I/O operation.

Link Control Block

The link control block (BLCB) is the primary control block for the BSC IOM for 3270 emulation. It is allocated in machine-wide storage when the BSC IOM for 3270 emulation task is created and exists until the task is destroyed. The BLCB contains the following data and control areas:

- Feedback record parameter area
- Pointers to other objects and control areas
- Status flags and counters
- Link control characters (EBCDIC)
- Work areas for the various BSC IOM for 3270 emulation routines
- Operation request elements (ORE)
- Program operation blocks (POB)
- Function operation blocks (FOB)
- Message operation blocks (MOB)
- Error and timer messages

Service Order Table

The service order table (SOT) is the secondary control block for the BSC IOM for 3270 emulation. It is allocated in machine-wide storage at LUD vary-on time. One SOT exists for each LUD that is varied on. The SOT contains information related to one session such as the request I/O hold queue used during active sessions, a copy of pertinent attributes of the device from the logical unit description, and the logical unit description session status. For 3270 operations, up to 32 LUDs can exist.

Service Order Table Address Table

The SOT address table contains pointers to the 32 SOT entries. The index received by the BSC IOM for 3270 emulation determines the session for which the message is intended.

Poll/Select List

The poll/select list enables the BSC IOM for 3270 emulation to communicate with the IOC to define the proper action the IOC should take to specific polls, general polls, or selects received for the host. The IOC automatically terminates a general or specific poll if the poll/select list indicates that the poll should be terminated. All selects received are returned to the IOM.

Session Line Buffer

One buffer exists for each session. This buffer is used to contain received data from the host.

STRUCTURE

The following is a list of the modules in 3270 emulation management and the function that each module performs. The list also shows how the module is invoked.

#TEBBIOM BSC 3270 Emulation Mainline Module

Function: Activates, manages, and de-activates the BSC telecommunication link communicating as a 3270 emulator to a host system.

How Invoked: Other VMC component.

#TEBELSE BSC 3270 Emulation Nonmainline Paths

Function: Performs all activity on the BSC telecommunication link not performed by #TEBBIOM.

How Invoked: Within this component.

#TEBERPL BSC 3270 Emulation Management Error Recovery Procedure for Link Processor

Function: Processes not hex 0100 BSTAT values and channel errors returned to the BSC 3270 emulation manager.

How Invoked: Within this component.

#TEBLPER Link Protocol Error Recovery Procedures for BSC 3270 Emulation Management

Function: Inspects the contents of the receive buffer and initiates a recovery action based on the buffer contents.

How Invoked: Within this component.

#TEBMODP BSC 3270 Emulation IOM—Modify ND, Modify CD, and Modify LUD Processor

Function: Processes messages that are sent to the main BSC 3270 emulation manager module (#TEBBIOM) as a result of a Modify ND, Modify CD, or Modify LUD instruction.

How Invoked: Within this component.

#TEBDE BSC 3270 Emulation IOM Diagnostic Processor

Function: Provides routines for a BSC 3270 emulation, internal trap, and read data store.

How Invoked: Within this component.

#TEXLATE 3270 Emulation Translation Function

Function: Provides translation support for 3270 emulation function that will interface with BSC line protocol.

How Invoked: Request I/O instruction.

#TEXLANG 3270 Emulation Language Translator

Function: Provides language group translation support for the 3270 emulation function.

How Invoked: Within this component.



X.25 Communications I/O Manager

INTRODUCTION

The X.25 Communications I/O Manager (XIOM) is a vertical microcode component that interfaces with the MSCP and up to 32 station IOMs, APPC/LU1 or remote work stations. It supports two instructions; Modify Network Description and Materialize Network Description. The XIOM activates, de-activates, and manages an X.25 data link attached to the System/38. There is one XIOM for each X.25 line.

The XIOM is created by the MSCP as the result of a Modify Network Description (Vary-on) instruction. The XIOM task is associated with one communications port or operation unit (OU) pair (transmit, receive) on the System/38. The XIOM destroys itself during the processing of a Modify Network Description (Vary-off) instruction. An overview of the XIOM is shown in Figure 34-1.

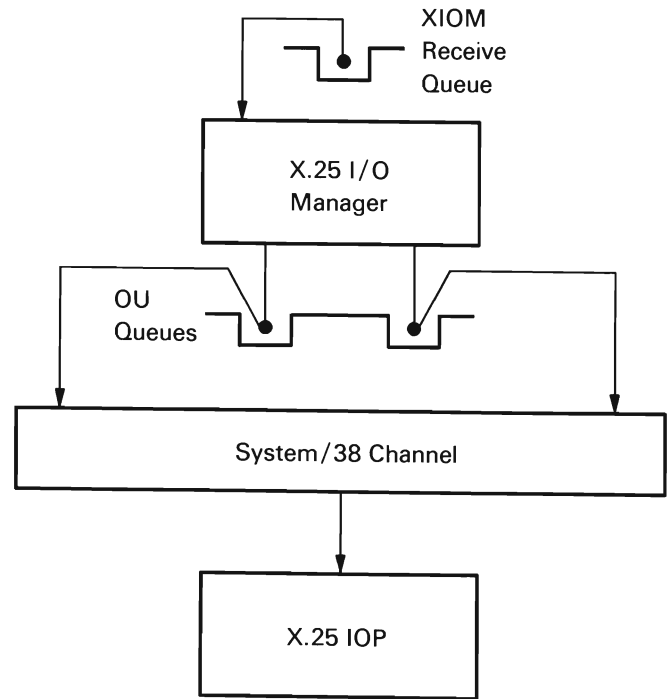


Figure 34-1. XIOM Overview

The integrated X.25 adapter provides a higher level of function than has traditionally been done in a System/38 line adapter. The adapter contains three layers of the open system interface (OSI) model: physical, data link control, and packet. IBM added a fourth layer, (QLLC, ELLC, or PSH) to provide adjacent node services such as XID, mode selection, SNRM/SABM, test, and disconnect. ELLC, also provides end-to-end confirmation.

Within the System/38 processing unit, the changes for X.25 support are mainly confined to the vertical microcode (VMC). The XIOM is between the existing SNA station IOMs and the integrated X.25 adapter. All SNA station IOMs can run on an X.25 network, and up to 32 stations can run through the XIOM concurrently. Each System/38 can support two integrated X.25 adapters concurrently, so using X.25 a maximum of 64 SNA stations can be running on the system concurrently. An overview of the System/38 implementation of X.25 architecture is shown in Figure 34-2.

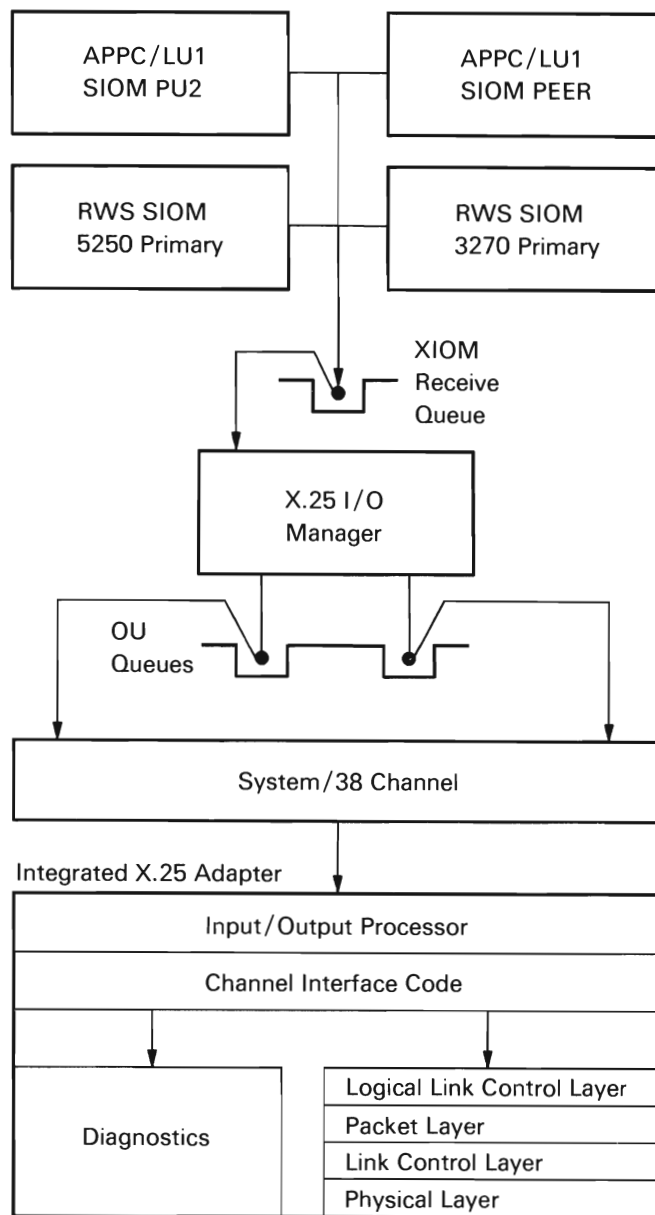


Figure 34-2. X.25 System Overview

DATA AREAS

Link Control Block (LCB)

The major XIOM control block is the link control block (LCB). The LCB contains all the information associated with the X.25 data link to the data communications equipment and to the pointers to tables that contain information for the control blocks for individual logical links.

The logical link control blocks are called service order tables (SOTs). There is one SOT for each active logical link, and a pointer to the current SOT is kept in the LCB whenever one is identified. The current SOT field in the LCB is important because it points to the active logical link control block.

All of the other control blocks in the XIOM are based on the structure of the LCB. This allows the control blocks to reside in machine-wide storage, which provides better readability of dumps and better performance. Figure 34-3 shows an overview of the LCB and associated control blocks and tables.

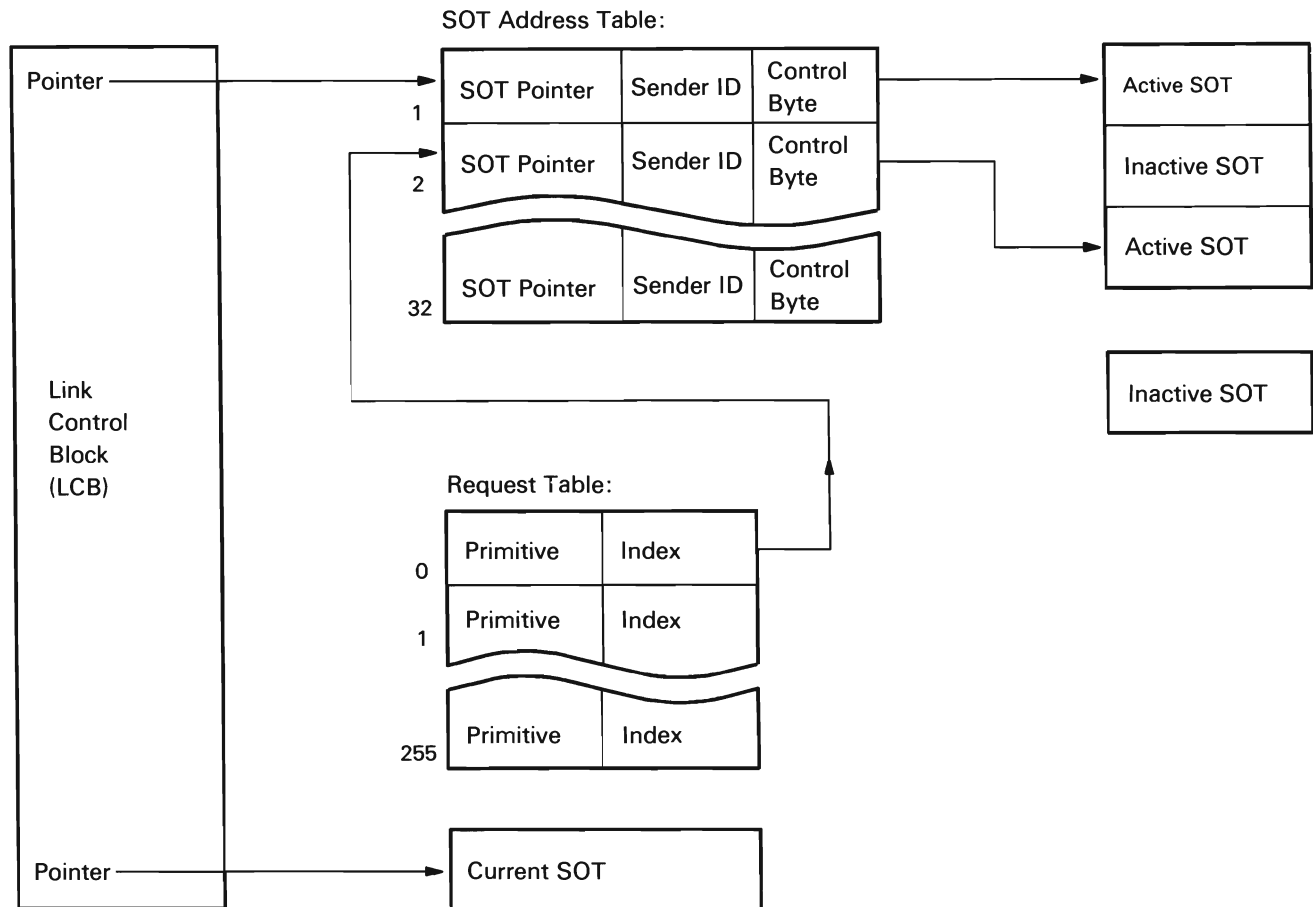


Figure 34-3. Overview of Link Control Block

Service Order Table (SOT)

The SOT contains all of the information necessary for each logical link or station. There is one SOT for each active logical link and each logical link is associated with one station.

The SOT contains queue headers, which the IOM uses to hold output requests from the stations that are in various states. It also contains pointers to CD oriented SRMs, partially completed input requests, and in general, all information pertaining to the communications link.

SOT Address Table

The SOT address table is the method used to index the SOTs. Since the table is relatively small, it can be kept resident, and can be referenced to obtain the active SOTs.

The SOT address table contains a pointer to an SOT, a VMC sender ID, and a control byte. The VMC sender ID routes output requests to the proper SOT.

Request Table

The request table is used to route confirmations to the proper routine as well as identify the proper SOT. The request table contains the primitive of the request being sent out, and the index of the SOT address table entry associated with the SOT for this logical link.

When a routine sends a request to the X.25 adapter, it makes an entry in the request table at the VMC ID index. VMC ID is a wraparound single-byte counter that is incremented each time a request is made. This allows 255 requests at one time.

Receive Buffers

The XIOM manages input buffers in a common pool for all routines. These buffers are a combination of:

- An input request send/receive message (SRM).
- A receive operation request element (ORE) structure.

A pool of buffers is built at ND vary-on time. Buffers are managed from common routines (#TXGBUFF and #TXFBUFF). The XIOM allocates more buffers if needed. Figure 34-4 shows an overview of the receive buffer structure.

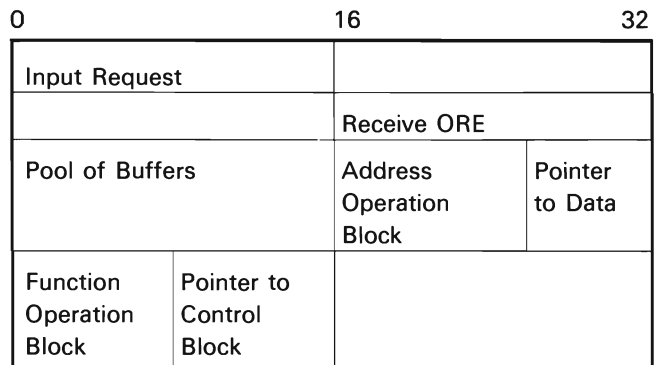


Figure 34-4. Receive Buffer

The address operation block (AOB) contains an address to an preacquired data buffer, and the function operation block (FOB) contains a pointer to a preacquired OSI block area.

Transmit Operation Request Elements (OREs)

The data transmit buffers are controlled by the SIOM. The XIOM maintains a single transmit ORE which contains enough AOB/FOB pairs to transmit the maximum amount of data that the adapter will allow to be outstanding. The XIOM also maintains a queue of transmit OREs which are shared by all routines.

Trap Table

The trap table contains a history of all CDs that ran on each logical link during the time the XIOM was active. The XIOM saves the CD name, the protocol (ELLC,QLLC, or PSH), and the character set (EBCDIC or ASCII) that the CD was using for the trap routines so those routines can go through the trap buffer and format the data correctly. The trap routines need to know the character set to interpret the data correctly. The routines also need to know what protocol was running in order to format the SNA data. The LLC header precedes the SNA data. Because each LLC protocol has a different length header, the LLC protocol must be known to increment the correct number of bytes past the header to get to the SNA data. When the XIOM is running, each logical link can support different CDs; therefore, the XIOM must keep a history of what CDs were supported by each logical link. This is done by using a link list that contains the CD name, protocol, and character set, as well as a pointer to the next node. Figure 34-5 shows the form of the trap table.

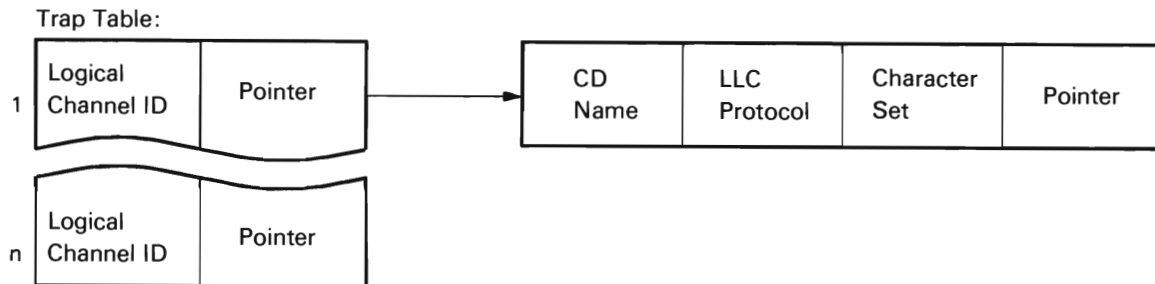


Figure 34-5. Trap Table

Valid Send/Receive Messages

The VMC SRM requests supported by the XIOM are listed below.

Message	Request	Response	Module	Origin
Start Trap	0000	8000	#TXTRAP0	DE
Start Trace	0001	8001	#TXTRAC0	CSM
Stop Trace	0002	8002	#TXTRAC0	CSM
Stop Trap	0010	8010	#TXTRAP0	DE
Contact	0201	8201	#TXCONTR	MSCP
Discontact	0202	8202	#TXDISCS	MSCP
Activate Link	020A	820A	#TXALK0	MSCP
Deactivate Link	020B	820B	#TXDACLN	MSCP
Connect Out	020E	820E	#TXCONOT	MSCP
Initialize Line	0215	8215	#TXINLN	MSCP
Abandon Connect Out	0218	8218	#TXABCON	MSCP
Request XID	0240	8240	#TXRQXID	MSCP
Inoperative Station	0281	NA	#TXINPST	SIOM
Channel Event Error	0300	8300	#TXCHAN	Channel
Channel Hardware Error	0400	8400	#TXCHAN	Channel
Channel Post Event Error	0500	8500	#TXCHAN	Channel
Channel Post Event Attention	0600	8600	#TXCHAN	Channel
Channel FOB Time-Out	0700	8700	#TXCHAN	Channel
Channel/OU Task Exception	0800	8800	#TXCHAN	Channel
Get Logical Channel Statistics	1011	9011	#TXSTAT1	SS
Get Logical Link Statistics	1012	9012	#TXSTAT2	SS
Second Time-Out	104F	904F	#TXLATE	MSCP
Output Request	2000	8000	TXOUTREQ	SIOM
Execute LLC Test	2005	A005	#TXLNK0	DE
Read Data Store	2007	A007	#TXRDS0	DE
Force SDR Update	3000	B000	#TXSTAT3	Error Log

STRUCTURE

The following list of modules in the XIOM includes the function that each module performs. This list also includes how the module is invoked.

#TXXIOM Mainline IOM

Function: Performs mainline data handling and routing for the IOM.

How Invoked: Other VMC components.

#TXNDUP Network Description Bring-up

Function: Brings up the X.25 Network Description object.

How Invoked: Within this component.

#TXCDUP Controller Description Bring-up

Function: Brings up the X.25 Controller Description object.

How Invoked: Within this component.

#TXDOWN Take-down Routines

Function: Takes down the X.25 CD and ND.

How Invoked: Within this component.

#TXERPL Error Recovery Routines

Function: This module contains all routines needed for error recovery.

How Invoked: Within this component.



Event Management

INTRODUCTION

Event management provides the function that allows the user to do the following:

- Monitor for user or machine signaled events
- Signal events
- Wait for the occurrence of an event

Event monitors are used to monitor for the occurrence of process-related and machine-wide events. These monitors are established when the Monitor Event instruction is executed. A monitor event template describes what to monitor and what action to take when the event is signaled. For process-related events, the monitor event template is inserted into the event monitor stack located either in the invocation work area (IWA) of the process or in an overflow area of machine-wide storage. For machine-wide events, an event entry is also inserted into the event management index.

Events are signaled either by the user by using the Signal Event instruction or by functions within VMC. When a machine-wide event is signaled, the event index is searched for processes that are monitoring the event. When a process-only event is signaled, the process being signaled is specified by a system pointer in the signal template. The event management then enqueues an event signaled message to the interrupt queue of the monitoring process.

When an event is signaled, more than one monitor can be signaled. Multiple monitors can be signaled in one process or in multiple processes for a machine-wide event.

Event management provides a function that allows a process to wait for the occurrence of an event. This is accomplished through the Wait-On-Event instruction. This allows the process that issues the instruction to remain in a wait state until the specified event is signaled or a specified time period has elapsed. When a time-out occurs, an exception is signaled to the monitoring process.

Event management supports the following System/38 instructions:

- Monitor Event
- Enable Event Monitor
- Disable Event Monitor
- Test Event
- Wait on Event
- Retrieve Event Data
- Cancel Event Monitor
- Signal Event
- Modify Process Event Mask

Monitor Event

#EMMNEVT is invoked as a result of a Monitor Event instruction. This module validates the event template and checks the event index for duplicate monitors. When the new monitor is validated, #EMMNEVT allocates space for the new monitor in the event monitor stack in the IWA (and the event management index if a machine-wide event) or in the machine-wide storage extension if the IWA is full.

Enable Event Monitor

#EMEBLED is invoked as a result of an Enable Event Monitor instruction. This module sets the enable/disable status bit to 0 (enabled). If the event monitor is in the signaled state, the disabled-schedule messages are enqueued to the event management queue in the process control block (PCB). If the specified event monitor is a timer event, #RMWTOE is invoked to establish the time interval. #CFESCH is then invoked to schedule any pending events.

Disable Event Monitor

#EMDBLED is invoked as a result of a Disable Event Monitor instruction. This module sets the enable/disable status bit in the event monitor to one (disabled).

Test Event

#EMTSEVT is invoked as a Test Event instruction. This module validates the specified monitor identification and then invokes #CFDQEMG to dequeue either the schedule-message that matches the specified monitor identification or the highest priority schedule-message if a monitor identification is not specified.

If a schedule-message is located, #EMREVTD is invoked to retrieve the event data, store the data in the area specified by operand 1, and set condition code to signaled. If a schedule message was not located, the condition code is set to not signaled. Inline code then performs a conditional branch when control is returned by #EMREVTD.

Wait-on-Event

The process can wait indefinitely for the event, or a specific wait time can be specified. The process remains in the wait state until the event is signaled or, if a time period is specified, until the time specified is reached and an exception is signaled to the process. If the wait is satisfied by an event and an event handler is not specified, #EMREVTD is invoked to move the event data to the area specified by operand 1.

Retrieve Event Data

#EMREVTD is invoked as a result of a Retrieve Event Data instruction. This module locates the event-related data using a pointer (set by #CFESCH, #EMTSEVT, or #EMWOE) in the PCB, and moves that data to the specified area.

Cancel Event Monitor

#EMCMEVT is invoked as a result of a Cancel Event Monitor instruction. This module locates the specified event monitor, marks the monitor as de-activated, and cancels the timer request if a timer event. If the event monitor is machine-wide, #EMCMEVT removes the entry from the event index and any pending events for the specified monitor are lost.

Signal Event

Figure 35-1 shows an overview to signal event. When an event is to be signaled, either by the machine or as a result of a Signal Event instruction, the event management signal and schedule event functions are invoked. These functions locate the process that is monitoring the event. For process-only events, the signal template contains a pointer to the process; for machine-wide events, the event index contains a pointer to the process and the event monitor with the process. Once the signal and schedule functions locate the event monitor, the function interrupts the monitoring process.

#EMSGEVT is invoked as a result of a Signal Event instruction. This module validates the instruction operands, copies the signal template to the IWA, and invokes #CFSGEVT.

#CFSGEVT is invoked for both VMC and user-signaled events. For machine-wide monitors, #CFSGEVT searches the event management index in three passes to locate a monitor. The first pass looks for a monitor with an identification that matches that of the event being signaled, the second for a matching generic subtype monitor, the third for a matching type subtype monitor. Each time a monitor is located, #CFSGEVT allocates and initializes a signal/schedule message and invokes #RMINIPI. #RMINIPI interrupts the signaled process and enqueues the signal to the interrupt queue of the signaled process. When a process-only event is signaled, #CFSGEVT then allocates and initializes a signal/schedule message, and invokes #RMINIPI to interrupt the signaled process and enqueue the signal message to the interrupt queue of the signaled process. (See *Process Interruption* in the *Resource Management* section of this manual for additional information concerning process interruption.)

When the process has been interrupted and becomes the highest priority task on the prime task dispatching element (TDE) queue, #CFESCH is invoked to schedule event handlers either at a System/38 instruction boundary (a maskable Supervisor Link instruction was issued) or when the process has been removed from a wait (for example, a wait-on-event). #CFESCH then invokes #CFDQEMG to remove the signal/schedule messages from the process interrupt queue (the rules specified in the monitor event template are enforced). When a message that contains event handler specifications is removed from the queue, a pointer to the message is put into the PCB (to allow a Retrieve Event Data instruction to retrieve the data) and the event handler is invoked (through #AICALLM). When all event handlers have been scheduled, #CFESCH returns control to the interrupted process at the next sequential instruction.

Modify Process Event Mask

This function is provided in translator generated in-line code. The current event mask can optionally be stored, and a new mask can be specified.

When the process is masked for events, the events are retained (to the limit specified in each monitor), but no event handlers are invoked.

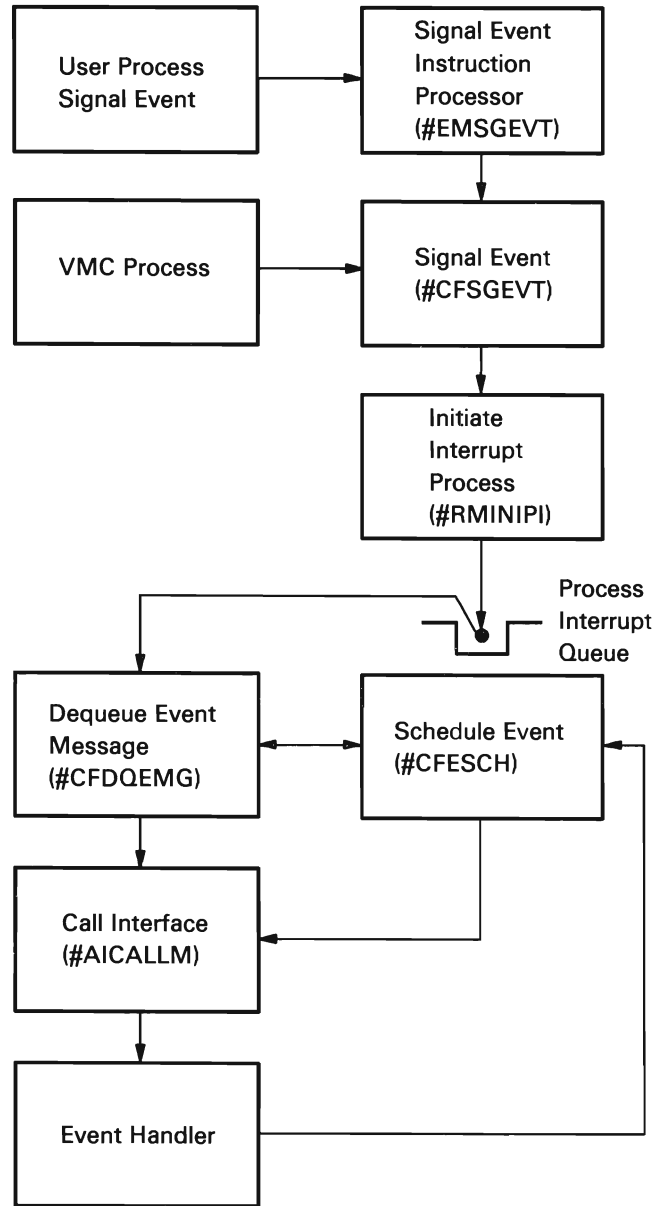


Figure 35-1. Signal Event Processing

Recovery

Component-specific exception handlers (CSEHs) are used to ensure the shared-exclusive lock mechanism (implemented by a send/receive counter) is not left in an unusable state when an exception occurs.

All areas modified by event management are allocated in temporary storage; therefore, damage determination and recovery for system termination is not required.

DATA AREAS

Monitor Event Template

The monitor event template describes the event to be monitored. The template contains:

- Template size and attributes
- A pointer to the event handler
- Event options
- Event compare value
- Event identification (class, type, subtype)

Event Index

The event index is a machine index initialized by process management at initial microprogram load (IMPL). The event index contains entries for all event monitors that monitor machine-wide event signals. The entries contain:

- Event identification
- Monitor event address
- Listener (process control space [PCS] address)

The event index is serialized by a shared-exclusive lock to synchronize all event management functions.

Signal Event Messages

The signal event messages are enqueued to the signal event queues. These messages describe the events to be scheduled.

Process Control Block (Nonresident)

The contents of the nonresident PCB that are specifically related to event management are:

- Event signal send/receive queues: Contain messages describing the events to be scheduled or signaled
- Pointer to the wait-on-event parameters
- Start of event related data chain
- Start of monitored event stack

Process Control Block (Resident)

The contents of the resident PCB specifically related to event management are:

- Process interrupt queue: Contains messages for the initial signal-event interrupt
- Process interrupt pending indicator: Indicates that the event-schedule function is active

Task Dispatching Element

The TDE contains the process-mask indicator used to mask processes from events, and the wait-on-event indicator used to indicate that the instruction executes within the process.

STRUCTURE

The following is a list of the modules in event management and the function that each module performs. The list also shows how the module is invoked.

#CFDQEMG Dequeue Event Messages

Function: Dequeues the event signal/schedule messages from the process interrupt queue and the event send/receive queues in the PCB.

How Invoked: Within this component.

#CFESCH Schedule Event

Function: Schedules the event handler if present in the monitor and ends wait-on-event when the desired event is signaled.

How Invoked: Other VMC components and within this component.

#CFSGEVT Signal Event

Function: Provides the common functions used to signal a machine- or user-sigaled event.

How Invoked: Other VMC components and within this component.

#EMDBLED Disable Event Monitor

Function: Disables the event monitor in the current process event stack.

How Invoked: Disable Event Monitor instruction.

#EMMNEVT Monitor Event

Function: Establishes a monitor within the executing process that detects the occurrence of the specified machine or user signaled event.

How Invoked: Monitor Event instruction.

#EMREVTD Retrieve Event Data

Function: Retrieves the event-related data and places the data in the specified space object.

How Invoked: Retrieve Event Data instruction and within this component.

#EMSGEVT Signal Event Instruction Processor

Function: Signals the specified event.

How Invoked: Signal Event instruction.

#EMTSEVT Test Event

Function: Provides synchronous testing for an event that may have been signaled within the process.

How Invoked: Test Event instruction.

#EMWOE Wait-On-Event

Function: Provides synchronization of the process with an external source by waiting for a specified event to occur.

How Invoked: Wait-On-Event instruction.



Exception Management

INTRODUCTION

Exception management provides both System/38 exception instruction support and exception handling. The System/38 instructions supported are as follows:

- Signal Exception
- Sense Exception Description
- Return From Exception
- Materialize Exception Description
- Modify Exception Description
- Retrieve Exception Data
- Test Exception

The Signal Exception instruction provides the capability of forcing a process into exception signaling mode. Once the exception is introduced into the system, it is processed similar to VMC-signaled exceptions. The Return From Exception instruction allows control flow to be redirected when processing of an exception is complete. The remaining instructions provide manipulation of exception data or exception descriptions.

Programs have the capability of setting invocation exit programs. These programs are executed when it is detected that the invocation that set the invocation exit is being bypassed. This occurs when control is given to a higher invocation to handle an exception from a Signal Exception instruction or from the exception generator or upon return from an exception handler (Return From Exception instruction). An invocation of an invocation exit program may not be bypassed.

Exception handling provides the following functions:

- Supports the Signal Exception instruction
- Supports VMC-signaled exceptions
- Intercepts both hardware-detected exceptions and machine checks
- Passes all exceptions to the machine interface

Exceptions originate from the following sources:

- The machine interface as a result of a Signal Exception instruction
- VMC as a result of VMC-detected errors
- Horizontal microcode (HMC) as a result of HMC-detected errors

Exception handling consists of routing and processing functions (see Figure 36-1) that allow VMC to attempt recovery from an exception before passing the exception to the machine interface. The exception management functions that perform the routing and processing functions are as follows:

- **Exception Generator:** The exception generator is the primary error handling function. All exceptions to be passed to the machine interface are directed to this function. The exception generator performs the following functions:
 - Searches for a user-defined exception handler
 - Prepares exception data
 - Performs invocation cleanup
 - Invokes the user-defined exception handler if one exists (otherwise the exception handler performs default action)
- **User-Defined Exception Handler:** These exception handlers are written using the System/38 instruction set and are used to process exceptions signaled to the machine interface.
- **Signal Exception Router:** Establishes the linkage between the generate exception Supervisor Link instruction (#SV1DEXC) and the third-level exception handler (TLEH) and invokes the TLEH.
- **TLEH (Third-Level Exception Handler):** The TLEH (#CFTLEH) is invoked from the machine check handler, the second-level exception handler (SLEH), or directly from VMC. The TLEH routes the exception to a component-specific exception handler (CSEH) and performs actions based on the status returned by the CSEH.

- CSEH (Component-Specific Exception Handler): A CSEH is defined by a VMC component to attempt to resolve an exception, replace the exception with a more meaningful one, or perform a cleanup function. A CSEH exists on a chain of exception handlers and is invoked by the TLEH.
- FLEH (First-Level Exception Handler): The FLEH (#SV00EXC) receives control from the HMC when the HMC detects an error. The FLEH then routes the exception to the appropriate common exception handler in VMC.
- Common Exception Handlers: These exception handlers process certain exceptions that occur because of an internal microprogramming exception encountered in either translator inline code or VMC code. The exceptions processed are those for which recovery is normally possible and as such are routed as quickly as possible to the common exception handlers from the resident FLEH.
- SLEH (Second-Level Exception Handler): The SLEH is a nonresident module that receives control directly from the FLEH for exceptions that do not qualify for a common exception handler. In addition, the SLEH may receive control from certain common exception handlers when an exception is unrecoverable. The SLEH then routes the exception to a default exception handler to process the exception.
- Default Exception Handlers: These exception handlers construct machine interface exceptions based on the occurrence of particular internal microprogram exceptions. This action is taken for internal microprogram exceptions which commonly occur during normal processing and are not considered errors in the VMC.

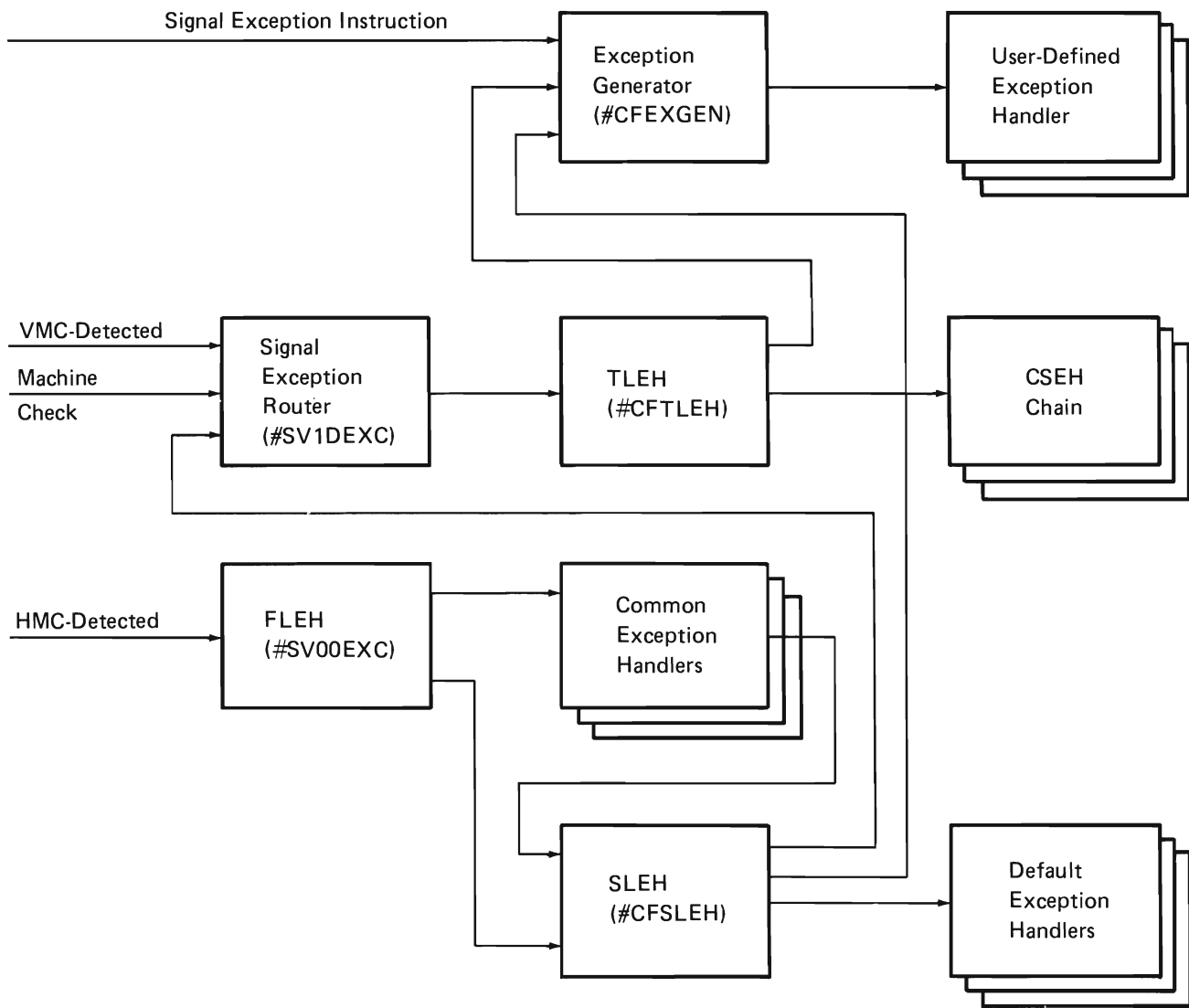


Figure 36-1. Exception Routine and Processing

First-Level Exception Handler

The FLEH (#SV00EXC) is given control from the HMC when an internal microprogramming (IMP) internal microprogramming machine exception occurs. The FLEH routes the exception to the appropriate common exception handler or the second-level exception handler as shown in the following list:

Time-out available call/return element (CRE) queue/CRE/task dispatching element (TDE) monitored	#RMTSACQ
Invalid description (02)	#CFSLEH
Busy (04)	#RMBSYXH
Allocate clear message (08)	#SMCLEXH
Monitored queue descriptor (0A)	#RMAEHS
Monitored message descriptor (0C)	#RMAEHS
Monitored TDE description (0E)	#RMAEHS
Send/receive count (SRC) overflow (10)	#CFSLEH
Address translation (12)	#SMPFEXH
Program event monitoring (14)	#RIPEMEX /#DOTRSEV
Execute (16)	#CFSLEH
Specification (18)	#CFSLEH
Addressing (1A)	#CFSLEH
Effective address overflow (1C)	#CFEAOSE /#CFEAOEH
Data (1E)	#CFSLEH
Binary overflow (20)	#CFSLEH
Binary divide (22)	#CFSLEH
Decimal overflow (24)	#CFSLEH
Decimal zero divide (26)	#CFSLEH
Operation (30)	#CFSLEH
Stack (32)	#CFSTKSE
Verify (34)	#CFSLEH
Chain conflict (36)	#CFSLEH
End of chain (38)	#CFSLEH
Edit digit count (3A)	#CFSLEH
Length conformance (3C)	#CFSLEH
Edit mask syntax (3E)	#CFSLEH
Invalid segment group address (40)	#CFSLEH
Second chain (46)	#CFSLEH

IMP exceptions can occur during the execution of a machine interface process or VMC task and can be caused by translator-generated inline code or VMC routines executing on behalf of the process. Some IMP exceptions are anticipated for the process, meaning that they can be caused by machine interface users and are a part of System/38 instruction support. Other IMP exceptions are unanticipated, meaning that they cannot be caused by the user or for proper execution of VMC routines. In this case, the exception is generally a result of an error in VMC code or damage and results in a machine check, function check, or a damage exception. The function of the common exception handlers are described in the following paragraphs. The common exception handlers are always active during execution of a process and are in support of several anticipated IMP exceptions.

Time Slice End and CRE Exception Handler

#RMTSACQ is invoked by the FLEH to process the following exceptions:

- Time slice end
- ACQ descriptor
- CRE descriptor
- TDE descriptor

When #RMTSACQ receives control from the FLEH, the address of the exception CRE is in register 2. The exception code in the CRE is examined, and #RMCREAC is called to replenish the CRE chain if the monitored CRE exception bit is on.

Then, the exception code is examined for the time slice end exception; if the bit is on, #RMREPTS is invoked to replenish the time slice value. All parameters and work storage needed to replenish the time slice value are in the main storage resident portion of the process control block (PCB) so that no page fault is incurred while replenishing the time slice. #RMTSACQ invokes #RMINIPI to interrupt the current process for the multiprogramming level (MPL) functions.

Time slice end is an asynchronous exception. It can happen during page fault handling; therefore, the monitored TDE exception is not unmasked until the current page fault is handled. A monitored CRE exception could be signaled because a page fault exception receives a monitored CRE. Therefore, #RMCREAC is main storage resident, does not page fault, and a special interface is established to get a main storage page frame for replenishing a CRE chain without causing a page fault.

Occurrences of monitored TDE exceptions are cumulated in static storage. The exception bits for TDE, CRE, and time slice end exceptions are set to zero before control is returned to #SV00EXC.

#RMTSACQ is main storage resident. It does not page fault and it does not use invocation work area (IWA).

Address Translation (Page Fault) Exception Handler

Routine #SMPFEXH receives control from the FLEH when an address translation exception (page fault) occurs. Page faults can occur for several reasons. The page fault exception handler processes any address presented for translation by an internal microprogram instruction or by an operation unit task.

Effective Address Overflow Exception Handler

#CFEAOSE is invoked by the FLEH as a result of an effective address overflow exception. Routine #CFEAOEH is invoked by #CFEAOSE if the effective address overflow exception was not a stack overflow using register 3. #CFEAOEH determines if the exception was caused by crossing a segment boundary within a segment group, and if so, completes the execution of the failing instruction. The failing program is then restarted at the instruction immediately following the failing instruction.

#CFEAOEH passes control directly to the SLEH if the internal microprogram instruction causing the exception is not one supported by the routine. Internal microprogram instructions *not supported* by this routine are as follows:

- ADD-ALHBL
- Branch
 - BAL
 - BALL
 - BC
 - BCN
 - BCNX
 - BCT
 - BU
- Compare and Swap Halfword
- Jump
 - JBF
 - JBN
 - JC

- System Control
 - DQM
 - DQTDE
 - DTDQ
 - EQM
 - EQTDE
 - FHR
 - FHRF
 - GHR
 - GHRF
 - HVVA
 - IPDE
 - RRCRR
 - RECC
 - RECM
 - SENDC
 - SENDM
 - SETCC
 - SETIT
 - SETTOD
 - STCC
 - STIT
 - STTOD

An effective address overflow exception generated by any of the preceding instructions causes the SLEH to gain control, CSEHs to process, and the default action of machine check to occur. The following list of internal microprogram instructions are supported and are processed by #CFEAOEH:

- Add
 - AC
 - AH
 - AHI
 - ALB
 - ALC
 - ALH
 - ALHI
 - AP
- And
 - NB
 - NBI
 - NC
 - NH

- Branch
 - TMBIBZ
 - TMBIBO
- Compare
 - CC
 - CH
 - CHI
 - CLB
 - CLBI
 - CLC
 - CLCL
 - CLCR
 - CLH
 - CLHI
 - CP
- Address Computation
 - CAL
 - CALH
 - CSA
 - CSAC
 - CSACH

- Convert
 - CVBP
 - CVPB
 - CVPZ
 - CVZP
- Divide
 - DHS
 - DP
 - DWS
- Edit
 - EDPD
- Exclusive OR
 - XB
 - XBI
 - XC
 - XH
- Execute
 - EX

- Load
 - L
 - LA
 - LB
 - LH
 - LM
 - LMB
 - LMH

- Move
 - MVBI
 - MVBIP
 - MVC
 - MVCL
 - MVCR
 - MVHI
 - MVNN
 - MVNZ
 - MVPS
 - MVZN
 - MVZZ

- Multiply
 - MHS
 - MP
 - MWS

- OR
 - OB
 - OBI
 - OC
 - OH

- Shift
 - SLHCT

- Store
 - ST
 - STB
 - STH
 - STM
 - STMB
 - STMH

- Subtract
 - SC
 - SH
 - SLB
 - SLC
 - SLH
 - SP

- System Control
 - CALLI
 - FNC2
 - RRCRR
 - LPDEA
 - LHTEA
 - EPDE
 - RPDE
 - IPDE
 - AHSP01
 - AHSP0
 - AFSP0
 - SVL1
 - SVL2
 - LSOP
 - LVT
 - STST
 - MVMAT
 - MVMAT
 - EXTAG
 - INTAG

- Test
 - TMBI
 - TMBIBZ
 - TMBIBO

- Translate
 - TR
 - TRT
 - TRR

- Trim
 - TRIM

- Zero and Add
 - ZAC

The following implicit SVLs are also supported:

- MPL
- DPL
- PPR

This routine locates the failing instruction using register BO and the instruction address register (IAR) stored in the CRE (the instruction length counter is subtracted from the IAR). The cause of the effective address overflow exception is determined by examining each operand base-displacement by locating the operand base register in the CRE, adding it to the displacement in a work area, and checking to see if a new segment was generated as part of the resulting address. Besides checking each operand, the data is checked to determine if it spans a segment boundary. The effective address, computed from each operand and added to the explicit or implicit length of the data addressed by the operand, is tested to determine if a segment boundary is crossed. All possible effective address overflow conditions for the instruction are tested and detected because several conditions can occur simultaneously. When multiple effective address overflow conditions occur, all are handled by this routine at one time to prevent recursions. If an effective address overflow recursion is detected, control is immediately passed to the SLEH.

When all the effective address overflow conditions have been detected, the failing instruction is moved to a work area and modified to compensate for the conditions. Each base-displacement operand is replaced by a work register number for the base and 0 for the displacement. The work registers are loaded with the effective addresses from the previous calculations. Any remaining register operands are assigned work registers, which are loaded with the contents of the corresponding operand-designated register stored in the CRE. If the data spans a segment boundary, then the data is moved to a work area and the appropriate operand work register is updated to point to the work area.

Once the instruction has been rebuilt, it is executed from the work area (execute instruction target). Since the effective address overflow condition is detected by the HMC prior to many other exception conditions, the instruction could fail again with a different type exception. An example is an arithmetic exception such as binary overflow or decimal zero divide. If any of these exceptions occur while executing the rebuilt instruction, the CRE and original storage areas are updated to reflect the results of the exception. The exception data is passed by #CFEAOEH to the second-level exception handler so that the normal action can be taken as described for the SLEH which includes passing control to CSEHs to process the exception.

After the instruction has been executed successfully, the post-execution environment must be captured and passed back to the user. The condition code set by the instruction will be captured and moved to the CRE containing the failing instructions' status. If a register was modified by the instruction, it is copied to the appropriate register save area entry in the CRE. Any data areas that were moved to work areas and then modified by the instruction will be moved back to their original location, taking care not to cause another exception. This routine then issues a Supervisor Exit instruction, causing the machine interface process to begin executing at the next sequential instruction, with the failing instruction now appearing to have completed successfully.

Some instructions supported by this routine require special considerations because of unique problems they present. Some of the known problems of this type are as follows.

Execute Instruction: If the failing instruction appears to be the Execute instruction, it is tested to determine if it failed with the effective address overflow exception, or if the instruction being indirectly executed failed. If the Execute instruction itself failed, control is passed directly to the SLEH (this is considered an error). Otherwise, the address of the instruction being indirectly executed is calculated from the decoded Execute instruction operand, and that instruction is processed as described previously.

Load/Store Multiple (LM, LMB, LMH, STM, STMB, STMH): Since the load/store multiple instructions can require the use of more work registers than are available, these instructions are simulated by moving the data directly to register save areas in the CRE or the reverse. These instructions do not change the condition code, so it can be left as it was in the CRE. Care is taken to compensate for the effective address overflow conditions during the moves. The other load and store instructions (L, LB, LH, ST, STB, STH) are treated as a subset case of these instructions, and processed the same way.

Move/Compare Characters Long (MVCL, CLCL): If the failing instruction is MVCL or CLCL, and the effective address overflow condition is caused by the address of either 8-byte operand control field, then control is passed directly to the SLEH (this is considered an error). Otherwise, the condition must have been caused by data spanning a segment boundary, and the current status of the operation is contained in the 8-byte control fields. These instructions are simulated, handling small pieces at a time until the segment boundary is crossed. Both operands could span segment boundaries in different relative locations. Also, the two operands can be of different lengths, requiring the pad character to be used. The 8-byte control field is updated to reflect the result of the operation, and for the CLCL instruction, a condition code is determined and returned in the CRE.

Compute Subscript Address (CSA): In addition to base-displacement, there are two other ways this instruction can cause an effective address overflow. The two conditions that may cause an effective address overflow are (1) if the index times the element size crosses the 64 K boundary, and (2) if the beginning address of the array plus the index times the element size crosses the 64 K boundary. In order to prevent recursive effective address overflows, the entire instruction is simulated rather than being reexecuted.

Tag Instructions (LVT, STST, MVCAT, MVASt, EXTAg, INTAg): These instructions require special handling to ensure that the tags are set correctly after reexecution. In many cases the operands must be quadwords and quadword-aligned. This ensures that they do not cross segment boundaries. In these cases the instruction can be reexecuted after the EA computation. The instructions of this type are LVT, STST, and MVASt. The MVCAT is executed in parts much like MVCL and CLCL in order to preserve the boundary alignment of the operands. The EXTAg and INTAg require that the operands be quadword-aligned. Therefore, they can be reexecuted using a quadword aligned work area by using MVCAT instructions for the data movement.

Supervisor Link Instruction (Supervisor Link 1, Supervisor Link 2): These instructions are reexecuted by simulating the hardware routing function. The exception code in the CRE is reset and the CRE reused for the supervisor link (SVL) CRE. The operand address(es) is computed and loaded into the proper register(s). The remaining registers are restored from the CRE and control passed to the proper SVL routine.

Implicit SVLs: These instructions are handled as regular internal microprogram instructions; however, only base/displacement operands are handled. For example, the implied parameters of the Edit instruction are not handled.

Busy Exception Handling

All busy exceptions are routed to #RMBSYXH by the FLEH. #RMBSYXH is an entry point in #RMBSYX that is nucleus-resident and contains executable instructions and a data area.

When #RMBSYXH receives control, it first performs a receive count to an SRC used to serialize data area access. Then it checks to determine if there is currently a busy waiter. If there is no current waiter, the interval timer is set for 250 msec. Then the count of waiters is incremented, a send count is performed to the serializing SRC, and a receive count is performed to an SRC where the TDE waits. When the wait ends, a Supervisor Exit instruction is performed to retry the original instruction causing the exception.

When the interval timer performs a send count, the task waiting for the timer SRC begins execution. That task performs an RECC to the SRC used to serialize access to the busy wait data area. Each TDE on the wait SRC is dequeued, its IAR is advanced to the Supervisor Exit instruction following the RECC, and the TDE is placed on the prime TDQ. Finally, an SENDC is performed to the SRC used to serialize data access. The interval timer task performs another Receive Count instruction to the interval timer SRC to wait until the next interval expires.

Access Exception Handler

Internal microprogram operations that access a queue can cause the following:

- Monitored queue access
- Monitored message access
- Monitored TDE access

#RMAEHS handles these monitored exceptions. Additionally, when queue operations are issued against the ACQ, IMP treats the ACQ as a send/receive queue (SRQ) for those operations and the same three access exceptions can be signaled.

#RMAEHS is invoked by the FLEH when one of the three access exceptions is signaled. These exceptions are encoded in exception code (byte 1) in the exception CRE, and only one exception is presented at a time. When control is transferred to #RMAEHS, the address of the exception CRE is in register 2.

#RMAEHS uses registers as work storage to decode the instruction that caused the access exception. Based on exception code and op code, it invokes #RMTDEAH (TDE access exception handler) or #PMSRQAC (queue access exception handler for queue support), and depending on the monitored bit in a message (or CRE), #RMAEHS either invokes #RMCREAC to replenish CRE or #PMSRMAC to support queue management.

#RMAEHS clears the exception code and issues a Supervisor Exit instruction to return control to the task dispatching function.

Second-Level Exception Handler

The SLEH (#CFSLEH) is invoked by the FLEH to process the following exceptions:

- Invalid descriptor (02)
- SRC overflow (10)
- Execute (16)
- Specification (18)¹
- Effective address overflow (1C)¹
- Data (1E)
- Binary overflow (20)
- Binary divide (22)
- Decimal overflow (24)
- Decimal zero divide (26)
- Operation (30)
- Verify (34)¹
- Chain conflict (36)¹
- End-of-chain (38)¹
- Edit digit count (3A)
- Length conformance (3C)
- Edit mask syntax (3E)
- Invalid segment group address (40)¹
- Second chain (46)
- Segment identification (SID) does not exist (80)¹
- Page not allocated (81)¹
- Permanent I/O error (82)¹
- Invalid pool state (83)
- Invalid pin request (84)
- Invalid write request (85)
- Bad main store page frame (86)¹

¹These exceptions are passed to default exception handlers.

The SLEH invokes the TLEH to process VMC exceptions, invokes the exception generator to process translated code exceptions, or invokes a default exception handler to process the exception.

The default exception handlers are described in the following paragraphs.

VMC Default Exception Handler

Routine #CFMIDEH is an exception handler that receives control from the SLEH as a result of a segment or page not allocated, effective address overflow, sector read error, or a bad main store page frame exception. It interprets the exception and generates an appropriate machine interface exception. The exceptions generated are as follows:

- Invalid space reference
- Object destroyed
- Space addressing violation
- Object suspended
- Parameter reference violation
- Segment header damage (machine check)
- Machine context damaged
- System object damaged (full and partial)

If the conditions cannot be mapped to one of these exceptions, the exception not handled is set and control is returned to the SLEH. CSEHs and machine interface exception handlers then get control.

#CFMIDEH receives control if a soft addressing exception (80 to 81), sector read error (82), bad main store page frame (86), or effective address overflow exception (1C) occurs and is not handled by the FLEH. The effective address overflow exception is normally handled by another routine (#CFEAOEH) from the FLEH, but in cases where a segment group boundary violation is detected, the exception is passed through the exception process to #CFMIDEH. If the exception is for a machine interface process, the failing address base (before overflow) is passed to the SLEH in the CSEH parameter interface. (If a 1C exception was not handled and was not a segment group overflow, the failing virtual address is set to 0; #CFMIDEH does not handle the exception.) #CFMIDEH then processes the effective address overflow exception exactly as the 81 exception because both imply an extent violation for some object. The processing of #CFMIDEH is generally as described in the following paragraphs.

Exception 80: This exception is processed as follows:

- If the internal microprogram exception code is 80, indicating that the failing address points to a segment that does not exist and is not part of an existing segment group, and for which the segment portion of the address is in the virtual-equals-virtual range, then a parameter list is generated for the object destroyed exception. The address of the list is placed in the parameter list supplied by the SLEH, and the return code is set to 4, indicating to the SLEH that a machine interface exception is to be issued.
- If the internal microprogram exception is 80 and the segment portion of the failing address is 0000FD00, then a parameter list is generated for the space addressing violation exception. The return code is set to 4 before returning to the SLEH.

- If the IMP exception is 80 and the segment portion of the failing address is 0000FE00, then a parameter list is generated for the parameter reference violation exception. The return code is set to 4 before returning to the SLEH.
- If the IMP exception is 80 and the segment portion of the failing address is 0000FF00, then a parameter list is generated for the invalid space reference exception. The return code is set to 4 before returning to the SLEH.

Exceptions 81 and 1C: These exceptions are processed as follows:

- If the 81 or 1C exception occurred and the object header indicates the object is suspended, a parameter list is generated for the object suspended exception, and the return code is set to 4 before returning to the SLEH.
- If the object suspended condition is not reflected in the header of the object, and if the error occurred in the space portion of an object, a parameter list is generated for the space addressing exception and the return code is set to 4 before returning to the SLEH.
- If none of the preceding conditions are met, based on the input address, then the return code is set to 0 in the SLEH parameter list indicating that this routine did not handle the exception. When the SLEH regains control from this routine, some system default action is taken.

Exception 82: This exception is processed as follows:

- If the 82 exception occurred and the address is the base page of a segment, the segment header damage machine check is signaled.
- If the bad page is not the base page of the segment, then the first page of the segment is found and used to locate the base segment of the object.

- If the object found is not a machine interface object, a sector read error on nonmachine interface object machine check is signaled.
- If the object is the machine context, machine context damage is signaled.
- If the object is a space object, journal space, or a data space, partial damage is set in order to allow the user to recover the rest of the data, and the partial object damage exception is signaled.
- Other machine interface objects have damage set which makes the object unusable, and the system object damaged exception is signaled.

Exception 86: This exception is processed as follows:

- The disk sector that corresponds to the bad main store page frame is found.
- The disk sector is marked logically bad to prevent use of the invalid data (this exception occurs only if the page has been changed).
- The exception is then handled the same as an 82 exception.

Verify Exception Handler

Routine #CFVFEH is a default exception handler that is used for the internal microprogram verify exception. #CFVFEH interprets the verify exception and either signals the appropriate machine interface exception or resolves the addressability of the late-bound pointer. If there is a late-bound pointer, #CFVFEH replaces (resolves) the pointer that caused the exception and exits in such a way that processing resumes with a retry of the operation that resulted in the exception.

The verify exception can only be caused by execution of the Load and Verify Tags internal microprogram instruction. Depending on the reason for the Load and Verify Tags instruction resulting in the verify exception, one of the following can result:

- A pointer does not exist exception (2401).
- An invalid pointer type exception (2402).
- Replacing of the pointer causing the exception with a resolved pointer. In an attempt to do this, other exceptions can result:
 - Object destroyed (2202): This can occur because the segment of the program that contains the symbolic name no longer exists or exists but has a new extender value. It can also result because the PSSA (process static storage area) space, the name resolution list (NRL) space, or a context has been destroyed.
 - Object not found (2201): This can occur because the symbolic name cannot be matched to a name in a context or to the name of an external data object.
- An object destroyed exception (2202) caused by a mismatch between the segment extender in the pointer and the one in the system object header.

It is also possible for the pointer to change in storage between the time of the execution of the Load and Verify Tags instruction and the time that this exception handler makes a copy of the pointer. Therefore, this function allows for the possibility of not being able to identify a cause of the verify exception; in this case it will exit and allow the excepting code to retry the Load and Verify Tags instruction.

#CFVFYEH receives control from the SLEH as a result of a verify exception. #CFVFYEH finds and decodes the Load and Verify Tags instruction using the IAR and instruction length count (ILC) fields of the CRE. Using the operand 2 effective address, it makes a copy of the pointer.

The pointer does not exist exception is generated if the Extract Tags internal microprogram instruction indicates that the referenced storage location does not contain a pointer.

If bit 2 of the pointer is equal to 1 (because the pointer is initialized but not yet resolved), the appropriate context management routine is invoked. Bytes 8 to 15 of an initial value pointer point to the symbolic name associated with the pointer. The symbolic name is in the external object list of the program that declared the pointer. The format of the name is: type, subtype, name, required authority. This name can be qualified with a context name for a system pointer or a program name for a data pointer.

Specification Default Exception Handler

Routine #CFSPECF processes certain internal microprogram specification exceptions, generally those that occur as a result of improper alignment on an operand. #CFSPECF signals the boundary alignment exception (0602) or the range violation exception (0603).

#CFSPECF is invoked by the SLEH as a result of a specification exception occurring in a machine interface process. The SLEH passes the address of a structure containing the excepting CRE address.

The IAR, base register 0, and ILC from the CRE are used to form the address of the excepting instruction. If the exception instruction is the Execute instruction, then the address of the target instruction is calculated.

Numeric Exception Handler

When an IMP binary or decimal overflow exception occurs while executing an encapsulated program, the numeric exception handler (#EXNUMEH) is invoked to determine if a size exception should be signaled immediately or if the exception should be temporarily ignored until the truncated result is put into the receiving field of the failing instruction. #EXNUMEH checks the ignore flag in the invocation control block (ICB) and sets the occurrence flag if the ignore flag is on. The ignore flag indicates that a potential overflow situation can exist in the work area, and the execution of the encapsulated program is to continue until the result is returned to the receiving area specified in the failing instruction.

Chain Conflict, End-of-Chain, and Second Chain Exception Handler

Routine #RMGFEX is a default exception handler for the chain exceptions generated during the execution of the following IMP instructions:

- Grant Hold Record
- Grant Hold Record First
- Free Hold Record
- Free Hold Record First
- Set Chain Busy

These instructions are used for seize/release, check lock state, and locking functions. The action performed by #RMGFEX is determined by the type of exception, as described in the following paragraphs.

Seize Conflict: #RMGFEX marks the conflicting hold record as monitored and enters the wait state by executing a receive count to an SRC in the TDE.

Lock Conflict: If a wait for the object is not indicated, #RMGFEX identifies the object in conflict and sets a return code that enables the excepting program to signal an exception to the machine interface.

If a wait is indicated, #RMGFEX identifies the conflicting object and marks the conflicting hold record as monitored. If the wait is asynchronous, processing continues as if the requested lock was obtained. If the wait is synchronous, a wait is entered by invoking #CFRWTO.

Check Lock Conflict: #RMGFEX identifies the conflicting object and sets a return code.

Monitored Release: #RMGFEX dispatches all TDEs waiting for the object by executing SENDC to the SRC of the waiting TDE.

Monitored Unlock: #RMGFEX locates all processes waiting to lock an object and sends a message to the lock wait queue for each waiter to restart each waiting process.

Third-Level Exception Handler

The TLEH (#CFTLEH) is invoked by the SLEH (through an SVL interface), the machine check handler, or the VMC directly. The TLEH consolidates the handling of HMC exceptions, VMC-detected exceptions, and hardware machine checks.

The main functions of the TLEH are as follows:

- Invokes VMC CSEH routines
- Performs an action based on the status of the exception after the CSEH routines have executed

CSEH routines intercept HMC exceptions, VMC-detected exceptions, and hardware machine checks caused in a routine of a VMC component or other routines in their invocation control block chain.

The entire exception handling mechanism has a dependency that VMC code executing as part of a machine interface process maintains base register 3 to be used only as a pointer into the IWA stack.

Invalid Segment Group Address Exception Handler

#CFINVSG is called by the SLEH when the invalid segment group address exception occurs in a user process. The SLEH passes the address of a structure containing the excepting CRE address.

The IAR, BPRO, and ILC from the CRE are used to form the address of the excepting instruction. If the excepting instruction is the Execute instruction, then the address of the target instruction is computed.

If the instruction causing the exception is the Branch Internal instruction and if the register defined by the first operand (B1) contains an address within a program system object, the branch target invalid exception is raised.

If the instruction causing the exception is the Compute Address Long or Compute Address Long Halfword instruction and if the register defined by the third operand (B3) contains an address within a system object or a related system object, the space addressing violation exception is raised.

If the instruction causing the exception is the Compute Subscript Address Constrained Halfword or Compute Subscript Address Constrained instruction, the space addressing violation exception is raised.

Establishing a CSEH

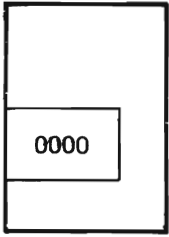
In order for the TLEH to route errors to a component-specific exception handler, the TLEH must be able to locate the address of the CSEH. The location of one or more CSEH routines is through a pointer in the current TDE to a chain of CSEH request blocks. If the pointer is 0, default exception processing is established. Otherwise, the pointer contains the address of an area normally in the invocation work area, that contains the address of a CSEH along with a chain pointer to the next CSEH, if one exists. An example of a CSEH chain is shown in Figure 36-2.

To establish a CSEH, VMC routines put a CSEH block at the beginning of the chain of active blocks for that task. Any number of CSEHs can be chained together, and chained in such a way that the last one chained is the first one processed. Each module within a process can have none or several exception handlers.

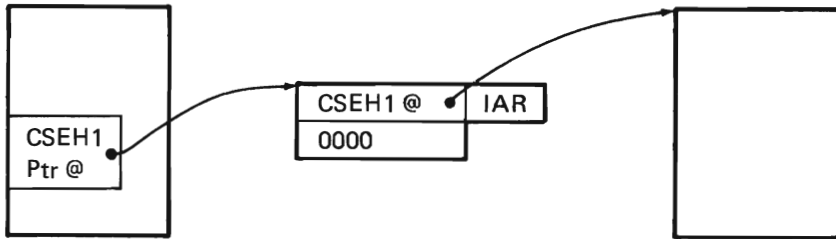
As each CSEH is invoked, it can process or ignore the exception. A return code indicates the action to be taken. If the exception was not handled, the TLEH then invokes the next CSEH if one exists.

Note: Each module that establishes a CSEH must also clear the CSEH before returning control to the caller. If a CSEH is not cleared, the pointers in the TDE to the CSEH block in the invocation can be pointing to an area that contains an invalid address.

TDE with No Active CSEHs



TDE with First CSEH Active



TDE with Second CSEH Active

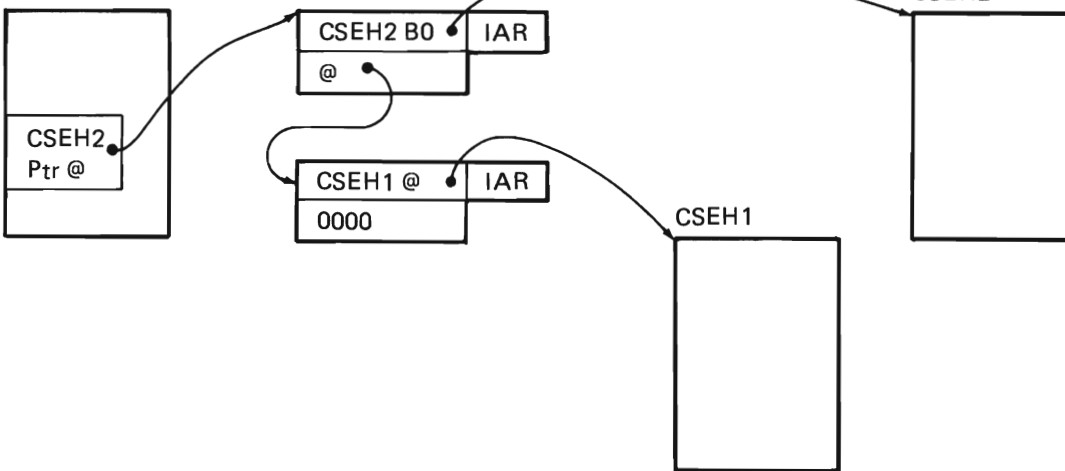


Figure 36-2. CSEH Chain

Damage CSEH

Routine #CFDAMEH is a common CSEH that handles unexpected exceptions and sets damage or provides tolerance. Damage can cause any exception type to be encountered but damage is inferred by the lack of any other exception handler for the exception. That is, an unexpected exception is assumed to indicate damage. This routine provides common actions for the following functions:

- Setting damage
- Tolerating damage (unexpected exceptions)
- Tolerating expected exceptions
- Linkage to VMC-specified cleanup routines

The damage exception handler provides facilities for setting damage (damage bit), generating damage exceptions and events, collecting logging information and causing it to be logged to the VMC log, invoking VMC-specified cleanup code, and returning control to specified locations (tolerance). For programs in which the action to be taken changes during instruction processing (including disabling of the exception handling), functions are provided to specify the action to be taken based on the range in the program in which the exception occurred.

This CSEH is established and receives control in the same manner as any CSEH. When #CFDAMEH receives control, it searches the location field of the action table to locate a match to the range in which the exception occurred. The IAR is used to do this. If a match is found, the options are checked to see if damage bit setting and damage exception generation (set option) are specified. The VMC cleanup routine (if any) is then invoked when control is returned to #CFDAMEH. #CFDAMEH branches to the point specified in the resume field. If no resume point is provided and set damage flag is not specified, then not-handled is assumed and the exception continues through the exception chain.

The CSEHDATA field in the CSEH request block area is used to hold information required to perform these functions. The ARRAYLBL field specifies the address of the action table to be used, the setting of B3 (saved in CSEH request block) is used to find the invocation control block and IAR for the program that defined the CSEH (not necessarily the same program as the one in which the exception occurred), and the object field is used to find the address of the object in which the damage bit is to be set.

DATA AREAS

CSEH Request Block

The CSEH request block is used to establish a CSEH routine and provide the linkage to that routine. The CSEH request block contains:

- A pointer to the CSEH routine
- An area used by the TLEH to build the CSEH chain
- Recursion information used by the TLEH
- User data information

STRUCTURE

The following is a list of the modules in exception management and the function that each module performs. This list shows also shows how the module is invoked.

#CFBGNEH Block Structured Exception Handler

Function: Records exception information and returns control to the routine that enabled the CSEH. From the recorded information, the routine can determine whether to tolerate the exception or resignal the exception so the routine can perform backout and cleanup functions.

How Invoked: As a CSEH.

#CFDAMEH Damage Exception Handler

Function: Processes exceptions caused by object damage, and provides tolerance to exceptions normally encountered during some operation.

How Invoked: As a CSEH.

#CFDAMST Damage Log and Set Routine

Function: Sets on damage bit in the encapsulated program architecture (EPA) header of a damaged object, optionally signals a damage set event, logs the damage, and optionally signals an object damaged exception.

How Invoked: Other VMC components or this component.

#CFDANGL Record and Tolerate Dangling CSEH

Function: Records information about an improperly enabled CSEH, disables all CSEH, and then signals a function check.

How Invoked: Within this component.

#CFDESEH Destroy a SID

Function: Destroys an SID if an exception occurs.

How Invoked: As a CSEH.

#CFEAOEH Effective Address Overflow Exception Handler

Function: Processes effective address overflow exceptions by determining if the exception was caused by crossing a segment boundary within a segment group, and if so, completes the execution of the failing instruction.

How Invoked: Within this component.

#CFEAOSE Attempt Recovery from Effective Address Overflow Exception

Function: Processes effective address overflow exception on a stack instruction.

How Invoked: Within this component.

#CFEXGEN Exception Generator

Function: Locates and invokes the exception handler (if one exists) for the signaled exception or performs default action.

How Invoked: Within this component.

#CFMIDEH Default Exception Handler

Function: Processes addressing and effective address overflow exceptions by mapping the HMC exception to another exception or sets an exception-not-handled condition.

How Invoked: Within this component.

#CFMSMEH CSEH for 80, 81, 82, and 86 Microprogramming Exceptions

Function: Traps 80, 81, 82, and 86 exceptions and returns the next microprogramming instruction.

How Invoked: As a CSEH.

#CFSLEH SLEH

Function: Determines the type of internal microprogram exception that occurred and passes control to the appropriate exception handler.

How Invoked: Within this component or other VMC components.

#CFSPECF Specification Default Exception Handler

Function: Processes exceptions for internal microprogram specification exceptions that occur as a result of improper operand alignment.

How Invoked: Within this component.

#CFTLEH TLEH

Function: Locates and invokes CSEH routines, and invokes the exception generator for machine checks that are not handled.

How Invoked: Within this component.

#CFTOLEH Tolerate a Specified Exception

Function: Resumes execution at a specified location if an exception of the type specified is detected.

How Invoked: As a CSEH.

#CFVfyEH Verify Exception Handler

Function: Interprets a verify exception and either signals the appropriate exception or resolves the addressability of a late-bound pointer.

How Invoked: Within this component.

#EXMDEXD Modify Exception Description

Function: Modifies the attributes of an exception description.

How Invoked: Modify Exception Description instruction.

#EXMTEXD Materialize Exception Description

Function: Materializes the attributes of an exception description.

How Invoked: Materialize Exception Description instruction.

#EXNUMEH Numeric Exception Handling Routine

Function: Processes execution-time numeric exceptions.

How Invoked: Within this component.

#EXRTEXD Retrieve Exception Data

Function: Retrieves the exception-related data and puts it into a specified area.

How Invoked: Retrieve Exception Data instruction.

#EXRTEXP Return From Exception

Function: Causes control to be passed from an exception handler to a specified instruction in an invocation within the process.

How Invoked: Return From Exception instruction.

#EXSGEXP Signal Exception

Function: Signals an exception to the process and (based on the results of the signal) optionally performs a branch to an instruction specified as one of the branch targets, or sets indicator targets to the appropriate value.

How Invoked: Signal Event instruction.

#EXSNEXD Sense Exception Description

Function: Searches a specified invocation of the process for an exception description that matches the specified exception and compare value, then returns the user data and exception handling action back to the process.

How Invoked: Sense Exception Description instruction.

#EXTSTEX Test Exception

Function: Determines if a specified exception description has been signaled and alters the control flow if the branch options are satisfied or sets indicators based on the result of the test.

How Invoked: Test Exception instruction.

#SV00EXC FLEH

Function: Determines the type of exception and routes control to the appropriate exception handler.

How Invoked: Entered from a direct microcode branch.

#SV1DEXC TLEH Router

Function: Routes control to the third-level exception handler.

How Invoked: Entered from a direct microcode branch.

Process Management

INTRODUCTION

Process management provides the functions required to accomplish a process, the basic unit of work in the system.

The areas that represent a process are shown in Figure 37-1. The task dispatching element (TDE) contains pointers to the process invocation work area and the process control block (PCB). The PCB contains pointers to the areas used by program execution management:

- The process automatic storage area
- The process static storage area
- The name resolution list (NRL)

The PCB is actually contained in two separate areas; one part is in the beginning of the process invocation work area, the remainder is appended to the TDE. The portion in the TDE contains information that must be resident in main storage to avoid a page fault when the associated process is not within the current multiprogramming level.

The fields required to initiate a process are specified in the process definition template (PDT). If the PDT specifies an access group, a pointer to the access group is maintained in the PCB. The access group can contain the process invocation work area, the process automatic storage area, the process static storage area, the NRL, and any other object used by the process.

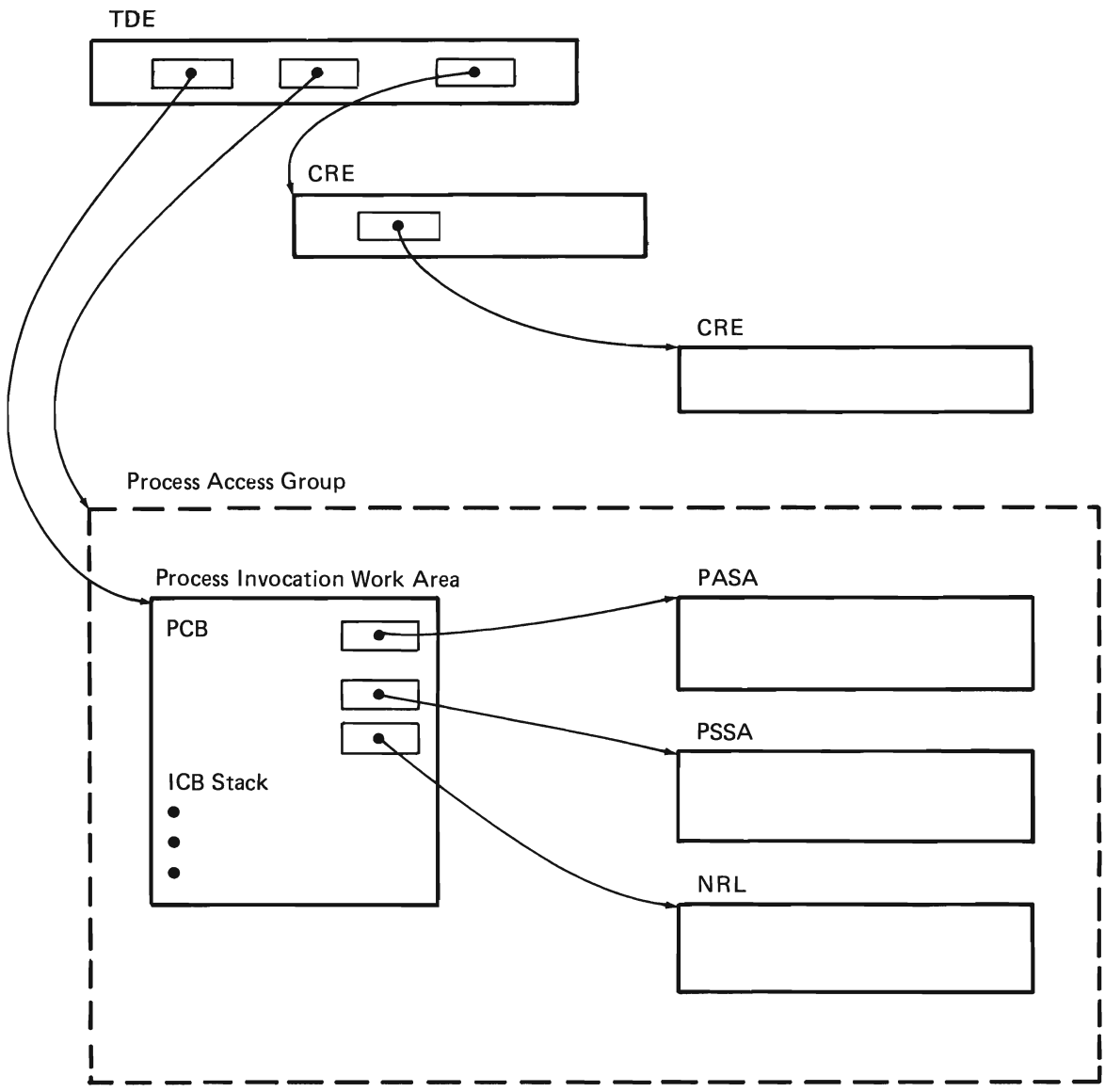


Figure 37-1. Process Management

Process management supports the following System/38 instructions:

- Create Process Control Space
- Destroy Process Control Space
- Initiate Process
- Materialize Process Attributes
- Modify Process Attributes
- Suspend Process
- Resume Process
- Terminate Instruction
- Terminate Process

Process management also supports functions used to create and destroy VMC tasks and microtasks.

Create Process Control Space

The Create Process Control Space instruction is used to create the work area process control space (PCS) required to support program execution. #PMCPCS performs the function.

A PCS is associated with a process as long as the process exists. This association begins just prior to entering the first machine instruction program (initiation or problem phase) and ends just after a process terminated event is sent from the service task (#RMSVTSK). A bit in the resident PCB of the PCS is set on during process initiation and is set off by terminate process. This bit indicates whether that PCS is associated with an active process.

Destroy Process Control Space

The Destroy Process Control Space instruction causes the specified PCS and addressability to it to be removed from a context (if addressed by a context). (The PCS to be destroyed must not currently be associated with a process.) Module #PMDPCS performs the destroy function.

Initiate Process

Process initiation is performed in two parts and executes under two TDEs.

Part 1 is performed by #PMINPR1 that is invoked by the supervisor link (SVL) router under the initiating process TDE. The basic functions performed in part 1 are as follows:

- Initial validation of the PDT is performed.
- Authority to issue privileged instruction is checked (#CFAUPRV).
- The PCB and the first invocation control block in the PIWA (second segment of the PCS) are initialized. If the new process is a dependent process, it is inserted into the subordinate process chain.
- A TDE is initialized (this includes pinning the first PCS segment containing the TDE and the resident PCB, removing from the machine storage pool chain, and chaining the TDE to the system TDE chain).
- The user profile is validated and seized (with code check) by #CFOCHKR, locked implicitly by #RMHLK, and transferred to the to-be-initialized process by #RMHXLK (a special transfer lock code that does not run under the receiving process).
- The new TDE is enqueued to the task dispatching queue (TDQ).
- The new TDQ is dispatched.

Part 2 of the process initialization is performed by #PMINPR2. This module running under the new TDE, completes validation of the PDT and invokes #RMINIPR to establish the process as part of the current multiprogramming level (MPL) and handle resource management process attributes (the process can enter the ineligible wait state at this point). #PMINPR2 then signals a process initiated event (#CFSGEVT) and invokes the initiation and problem phase programs. Finally, #PMINPR2 initiates process termination after normal completion of the specified programs by transferring control to #PMDPROC.

Materialize Process Attributes

The attributes of a process are materialized by module #PMMATER. This module can materialize either the attributes of both the current process or an external process. Machine-wide storage (MWS) is obtained, and PCB of the target process is copied to the MWS. This provides a copy of the attributes of a process at a specific point in time. The data is then moved from MWS to the receiving area.

Modify Process Attributes

The Modify Process Attributes instruction is performed in two parts. The modify function operates under two TDEs if an external process is being modified.

Part 1 is performed by #PMMODF1, which is invoked by the SVL router. This module performs validation and builds a message containing the modify option and the modify data. If #PMMODF1 is modifying itself, then it enqueues the message to the process interrupt data queue in the PCB. If an external process is being modified, #PMMODF1 enqueues the message to the interrupt process function (#RMINIPI) and then invokes that function to interrupt the target process. #RMINIPI invokes #PMMODF2 to execute under the TDE of the interrupted process. The modification data is passed to the target process through the process interrupt queue.

Part 2 is performed by #PMMODF2. #PMMODF2 dequeues the message from the process interrupt data queue and modifies the affected process attributes based on the data in the message. Because modification is asynchronous to the originating instruction, #PMMODF2 may not find any messages waiting or may find multiple messages waiting. If the attribute to be modified is related to resource management (for example time-slice interval or priority), the preceding sequence is followed, except that #RMMODAT is invoked rather than #PMMODF2. The module invoked is determined by the interrupt function code set by the interrupting process.

If the attribute to be modified is a user profile, the input user profile is validated and seized (with lock check) by #CFOCHKR. Next, an implicit lock on the user profile is obtained by invoking #RMHLK. Then if the target is not the current process, an implicit lock transfer is performed by #RMHIXLK. The target process invokes #PMMODF2 to complete the lock transfer and then invokes #RMHUNLK to release the profile implicit lock of the previous user.

Suspend Process

A process is suspended as a result of a Suspend Process instruction. This instruction causes module #PMSUSPR to be invoked. #PMSUSPR validates the input parameters. If subordinate processes are to be suspended, the #PMSUSPR invokes #PMRSUBS and the following steps are executed until all subordinate processes have been suspended:

1. #PMRSUBS locates (using the subordinate process chain) all subordinate processes associated with the target process and invokes #PMSPCIR for each subordinate process.
2. #PMSPCIR updates the external existence state in the resident PCB and invokes the process interrupt function (#RMINIPI) to interrupt the subordinate process and invoke #PMSPTAR under the process that is to be suspended.
3. #PMSPTAR updates the existence state in the PCB, signals a process suspended event, optionally sets access control bits (in the resident PCB), and invokes the receive with wait-time-out functions (#CFRWTO) to put the process in a wait state.

If the process being suspended is not the current process, #PMSUSPR invokes #PMSPCIR to invoke the interrupt function and invoke #PMSPTAR as in steps 2 and 3 for subordinate processes. If the current process suspends itself, #PMSUSPR invokes #PMSPTAR to suspend the target process as in step 3 for subordinate processes.

Terminate Instruction

Terminate instruction consists of the module `#PMTRMIN`. Other components, which have long running MI instructions, can check the terminate instruction interrupt bit. If this bit is on, the operation can be canceled.

`#PMTRMIN` is invoked by the supervisor link (SVL) router. It validates the input operands, obtains subordinate process chain lock, and performs an internal hold against the target process. It then checks for process control special authority if the current process is not the parent of the target process. The `ENABPINT` macro is used to interrupt the target process and to set the interruption bit in the resident process control block. Clean-up is performed by releasing the lock, and freeing the target process (`INTFREE`). If the target process is the current process, `#PMTRMIN` returns because there is no long-running function to terminate (`#PRTRMIN` is running). `#PMTTCSEH` is used as the exception handler for the module.

Resume Process

The execution of a process is resumed as a result of a Resume Process instruction. This instruction causes module `#PMRSMMPR` to be invoked. `#PMRSMMPR` validates the input parameters. If subordinate processes are to be resumed, `#PMRSMMPR` invokes `#PMRSUBS` and the following steps are executed until all subordinate processes have been resumed:

1. `#PMRSUBS` locates (using the subordinate process list) all subordinate processes associated with the target process and invokes `#PMRPCIR` for each subordinate process.
2. `#PMSPCIR` invokes the process interrupt function (`#RMINIPI`) to interrupt the subordinate process and invokes `#PMRPTAR` under the process that is to be resumed.
3. `#PMRPTAR` sends a message to the suspend process queue and returns through the interrupt function to the receive with wait-time-out function (`#CFRWTO`). The receive function executes a successful Receive Message instruction and returns control to `#PMSPTAR`.
4. If the subordinate process had suspended itself, `#PMSPTAR` returns control to `#PMSUSPR`. If the subordinate process was suspended by another process, `#PMSPTAR` returns control to the interrupt function and control is returned to the program at the point of interruption.

The target process is then resumed in the same manner as subordinate processes.

Terminate Process

Processes are terminated when one of the following occurs:

- Normal return: A process reaches the end of the problem phase.
- Exception termination: An exception occurs that is not handled by the process.
- Terminate instruction: A Terminate Process instruction is executed.
- Superordinate process termination: A process terminates as a result of a superordinate process terminating.

Figure 37-2 shows an overview of process termination.

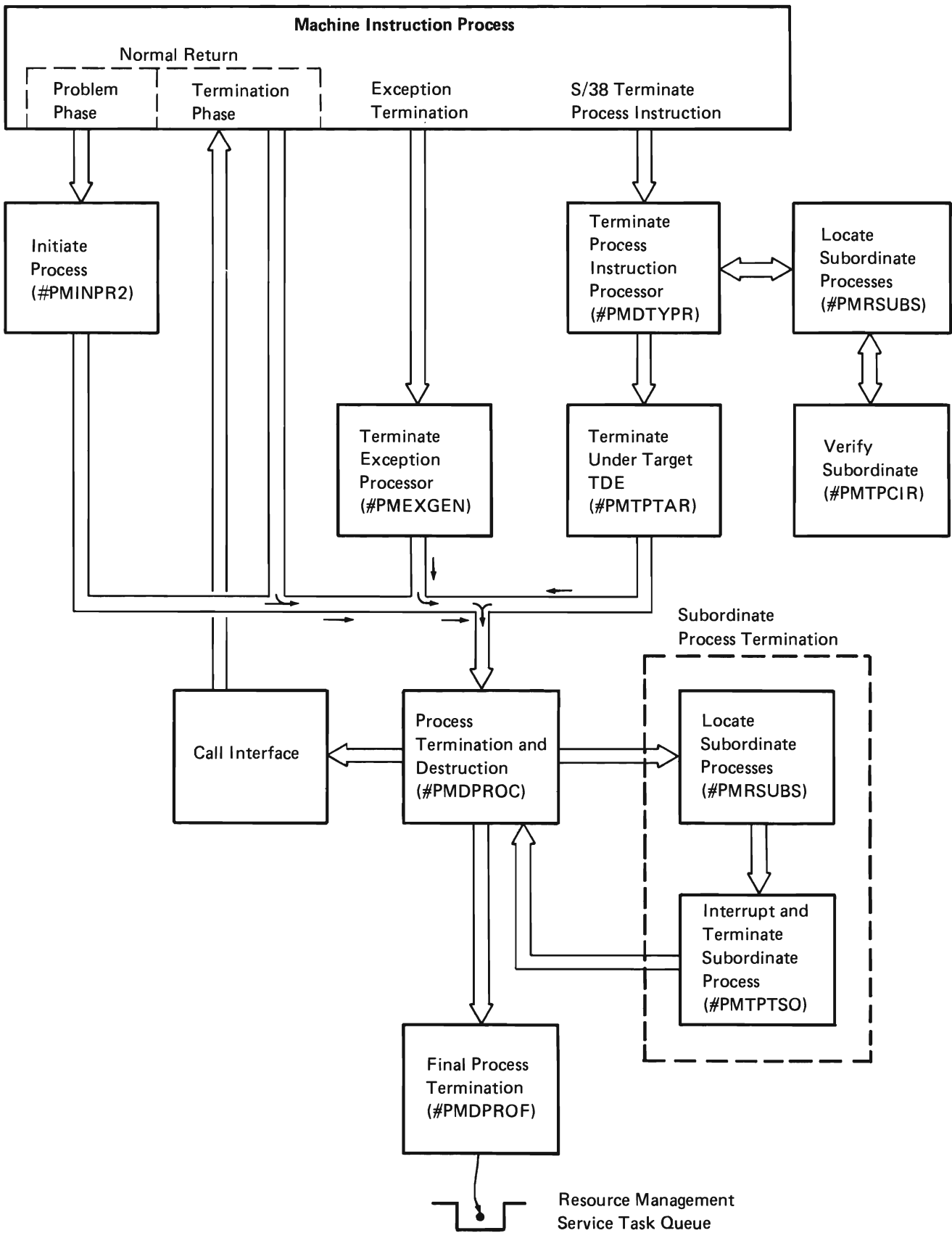


Figure 37-2. Process Termination Overview

Normal Return

Normal return from the problem phase of a program (or return from the initiation phase of a program if no problem phase is specified) causes #PMINPR2 to be invoked. #PMINPR2 sets the termination status in the PCB and invokes #PMDPROC as the highest invocation level. #PMDPROC then invokes the termination phase of the process and terminates the process.

Exception Termination

Exception termination occurs when an exception is detected and neither an exception handler nor the process default exception handler is specified to process the exception. In this case, #PMEXGEN is invoked by the exception generator. #PMEXGEN sets the terminate status in the PCB and invokes #PMDPROC to terminate the process in the same sequence as for a normal return. The process is destroyed if an unhandled exception occurs in the termination phase.

Terminate Instruction

The instruction to terminate a process can be issued either from an external process or from within the process to be terminated. The terminate process instruction processor (#PMDTYPR) is invoked by the SVL router as a result of a Terminate Process instruction. This module first validates the instruction and then begins terminating the process. If the process contains subordinate processes, #PMDTYPR invokes #PMRSUBS and #PMTPCIR to locate and terminate all associated subprocesses. This is accomplished through the process interrupt function causing #PMTPTAR to be invoked under the TDE of the subordinate processes.

Module #PMDTYPR then invokes #PMTPTAR. This module sets the termination status in the PCB and invokes #PMDPROC to complete termination of the process. #PMDPROC allows the subordinate processes to enter their termination phases. If a subordinate process does not complete the termination phases before the target process, the subordinate process is interrupted and terminated.

Subordinate Process Termination

When a target process is to be terminated, all processes subordinate to the target process are also terminated. If the target process terminates because of normal return or exception termination, subordinate processes are terminated by #PMDPROC and #PMTPTSO.

Create Task

The create task function #CFCTASK creates a VMC task (TDE and IWA [invocation work area]) or a microtask (TDE only). The input parameter list specifies the first module to be invoked under the new task, the amount of automatic storage required, the priority of the new process, and the number and content of the initial register values.

Destroy Task

The destroy task function (#CFDTASK) provides the functions that allow a module to destroy its own task or an external microtask.

Note: The microtask is assumed to be waiting on a queue from which the task can be removed without damaging the system.

DATA AREAS

Process Control Space

The PCS contains the work areas required to execute a program within a process. Figure 37-3 shows an overview of the contents of the PCS.

Resident Portion

Segment Group Header
EPA Header
Process Control Block (resident)
Task Dispatching Element

Pageable Portion

Segment Group Header (partial)
Process Control Block
Default Exception Description
Event Monitor(s)
Invocation Control Block
Invocation Control Block

Figure 37-3. Process Control Space

Process Definition Template

The PDT (ZZPDT) defines the attributes of a process to the initiate process function. Figure 37-4 shows an overview of the contents of the PDT.

Process Control Attributes
Resource Attributes
Modification Attributes
Process Pointers
<ul style="list-style-type: none">• User Profile• Process Communication Object• NRL• Initiation Phase Program• Termination Phase Program• Problem Phase Program• Process Default Exception Handler• PASA• PSSA• Process Access Group

Figure 37-4. Process Definition Template

STRUCTURE

The following is a list of the modules in process management and the function that each module performs. The list also shows how the module is invoked.

#PMPCPS Create Process Control Space

Function: Creates a process control space.

How Invoked: Create Process Control Space instruction.

#PMPCSC Create PCS Component-Specific Exception Handler (CSEH)

Function: Performs the required recovery for exceptions that occurred during a process control space creation.

How Invoked: Other VMC components.

#PMDPCS Destroy Process Control Space

Function: Destroys a PCS that is not currently associated with a process.

How Invoked: Destroy Process Control Space instruction.

#PMDPROC Process Termination Phase and Process Destroy

Function: Puts a process through the termination phase and then destroys the process. If the process has subordinate process, the subordinates are interrupted and destroyed.

How Invoked: By return from problem phase.

#PMDPROF Process Terminate Final Code

Function: Dequeues the current TDE from the TDQ and sends the terminate message to the service task.

How Invoked: Other VMC components.

#PMDTYPR Terminate Process Instruction

Function: Terminates a process or subordinate processes or both.

How Invoked: Terminate Process instruction.

#PMEXGEN Terminate Process Call from Exception Generator

Function: This module is invoked by the exception generator to initiate termination of the process. This invocation occurs only when a process has neither user exception handler nor a process default exception handler.

How Invoked: Other VMC components.

#PMFCSEH Modify Process Attributes CSEH

Function: Provides the required recovery from an exception that occurred during a modify process attributes instruction.

How Invoked: Other VMC components.

#PMFINAL Final Process Termination

Function: Final termination code for a process or a VMC task.

How Invoked: Other VMC components.

#PMICSE1 Initiate Process Part 1 CSEH

Function: Provides recovery for initiate process part 1.

How Invoked: Other VMC components.

#PMICSE2 Initiate Process Part 2 CSEH

Function: Provides recovery for initiate process part 2.

How Invoked: Other VMC components.

#PMINIT Process Management IPL Initialization

Function: Provides an initial program load (IPL) initialization for all process management functions.

How Invoked: Other VMC components.

#PMINPR1 Initiate Process Part 1

Function: Establishes a new process using the process control space specified and the attributes in the PDT.

How Invoked: Initiate Process instruction.

#PMINPR2 Initiate Process Part 2

Function: This module is invoked as the first invocation under a new TDE. This module completes initiation of the process and invokes the user initiation phase and/or the problem phase programs.

How Invoked: Task switch under TDE of the new process.

#PMIPL1 Build Process Environment for IPL
Sequence Part 1

Function: Builds the machine process.

How Invoked: Other VMC components.

#PMMATER Materialize Process Attributes

Function: Materializes one or all specific attribute(s) of a process.

How Invoked: Materialize Process Attributes instruction.

#PMMODF1 Modify Process Attributes

Function: Modifies a specific attribute of a process.

How Invoked: Modify Process Attributes instruction.

#PMMODF2 Modify Process Attributes Part 2

Function: This module executes under the TDE of the process being modified and makes the actual update of the PCB.

How Invoked: Other VMC components.

#PMRPCIR Resume Subordinate Process Initiation

Function: Initiates the resumption of subordinate processes of a process.

How Invoked: Other VMC components.

#PMRPTAR Resume Process Logic Under Target
TDE

Function: This module executes under the TDE of the process being resumed and sends a message to satisfy the suspend receive message.

How Invoked: Other VMC components.

#PMRSMR Resume Process

Function: Resumes a process and/or all of its subroutines.

How Invoked: Resume Process instruction.

#PMRSUBS Reference All Subordinate Processes in Chain

Function: References all subordinates of a given process and invokes a specified function against them.

How Invoked: Other VMC components.

#PMSPCIR Suspend Subordinate Process Initiation

Function: Initiates the interrupt function against all subordinate processes of a process.

How Invoked: Within this component.

#PMSPTAR Suspend Process Logic Under Target TDE

Function: This module executes under the TDE of the process being suspended and performs the suspend operation.

How Invoked: Other VMC components.

#PMSUSPR Suspend Process

Function: Suspends a user process and/or all of its subordinates.

How Invoked: Suspend Process instruction.

#PMT CSEH Terminate Process CSEH

Function: Performs the required recovery for exceptions that occur during process termination.

How Invoked: Other VMC components.

#PMTPCIR Terminate Process Verify Subordinate

Function: Verifies that this subordinate is not the process issuing the terminate instruction. This module is passed to #PMRSUBS and is called for every subordinate under the root process being terminated. This module executes under the process issuing the terminate instruction.

How Invoked: Within this component.

#PMTPCUP Remove Subordinate Process from Process Chain

Function: Locates a subordinate process and removes the process from the chain.

How Invoked: Within this component.

#PMTPTAR Terminate Process Logic Under Target TDE

Function: This module executes under the TDE of the process being terminated and initiates the termination of that process.

How Invoked: Other VMC components.

#PMTPTSO Interrupt and Terminate Subordinate
Process

Function: This module is invoked against all subordinate processes that complete termination and are being removed from the system.

How Invoked: Within this component.

#PMTRMIN Validate Input Operands

Function: Validates input operands, obtains subordinate process chain lock, and performs an internal hold against the target process.

How Invoked: SVL router.

#PMZCSEH Materialize Process CSEH

Function: Provides the required recovery from exceptions that occur during a materialize process attributes operation.

How Invoked: Other VMC components.

Resource Management

INTRODUCTION

Resource management consists of VMC functions that provide certain supervisory functions and support those System/38 instructions that control machine resources. The functions provided by resource management are as follows:

- Machine support: Routines that support the machine queuing and dispatching functions.
- Object serialization: Routines that support the System/38 locking instructions and the seize/release function.
- Timer services: Routines that provide a time-out function and support the time-of-day clock.
- Process interruption: A routine that provides a means to interrupt another process (at a System/38 instruction boundary, including entry into a wait state). This allows another function such as event scheduling, process termination, and lock transferring to be performed.
- Multiprogramming level (MPL) support: Routines that allow the user above the machine interface to control the levels of multiprogramming in the machine. The machine support, process interruption, and MPL routines are closely related.
- Resource management service task: Routines that provide a variety of functions required for the execution of asynchronous tasks.
- Resource management attribute control: Routines that support the System/38 instructions that materialize resource management data and modify resource management controls.
- Access group control: Routines that support the System/38 instructions that create, materialize, and destroy access groups.

Machine Support Functions

The machine support functions provide the following support:

- Handle access exceptions
- Maintain the availability of call/return elements (CREs) and task dispatching elements (TDEs)

Access Exceptions

There are three types of access exceptions associated with queuing operation. These exceptions are as follows:

- TDE
- Send/receive message (SRM)
- Send/receive queue (SRQ)

VMC uses these exceptions to detect when a process enters a wait state and to maintain the availability of CREs on the available CRE queue (ACQ). Normally these exceptions are presented in byte 1 of the exception code in the CRE. #SVOOEXC (the first-level exception handler) invokes #RMAEHS (resource management exception router). #RMAEHS then invokes the appropriate routine to process the exception. These routines and the exceptions they process are as follows:

- #RMTDEAH for TDE access exceptions

Note: TDE access exceptions are used only to indicate that a process has entered a wait state. These exceptions occur when a Receive Message instruction is executed in module #CFRWTO.

- #PMSRMAC (in queue management) for SRM access exceptions when the message is not a CRE
- #RMCREAC for SRM access exceptions when the message is a CRE
- #PMSRQAC (in queue management) for queue access exceptions

CRE Availability

Horizontal microcode (HMC) uses CREs to save registers during supervisor linkage (SVL) operations. If the TDE that issued an SVL does not have any free CREs available, HMC performs an implicit receive message to the ACQ. The operations in the following paragraphs are performed to ensure that a CRE is always available for use.

The fourth from the last CRE on the ACQ is marked as monitored. (Initially, a supply of CREs on the ACQ is assembled in nucleus module #RTTASKS.) When a TDE obtains a CRE from the ACQ by an SVL (explicit or implicit, including those used to present exceptions) and the CRE obtained is monitored, a CRE access exception is indicated in byte 0 of the CRE exception code, and the exception is signaled. #SVOOEXC invokes #RMTSACQ to determine that a monitored CRE exception has occurred and invokes #RMCREAC. This routine obtains a frame of main storage by invoking #SMALCPF. (This action requires no additional CREs because the modules involved do not cause any exceptions.) #RMCREAC then formats the frame obtained into four CREs and chains them to the ACQ, and sets the monitor indicator in the fourth from last CRE. (The monitor indicator in the CRE that caused the exception was turned off.) #RMCREAC then returns to #RMTSACQ which in turn invokes #SVOOEXC. This linkage sequence allows additional exception to be presented in byte 0 of a CRE causing an exception. #SVOOEXC continues to invoke #RMTSACQ until the additional exceptions are processed. #SVOOEXC then either performs a supervisor exit (SVX) or invokes the exception handler indicated by byte 1 of the exception code.

Logic in #RMCREAC attempts to ensure the success of the algorithm by marking the executing TDE as not timed (to prevent a timer exception that will require an additional CRE) and by checking that the executing task does not hold the MSM lock (in which case the TDE will deadlock after the call to #SMALCPF). A deadlock condition is possible only if a task holding the main storage management (MSM) lock incurred a timer exception and encountered the monitored CRE when presenting the exception. Storage management does not prevent this condition. When #RMCREAC is entered by a task holding the MSM lock, #SMALCPF is not called. Instead, a warning bit is set in the machine communications area, and a message is sent to the service task requesting that more CREs be created.

If a get CRE or receive message operation is performed and storage management obtains a monitored CRE, the exception is presented as an SRM access exception (in this case the ACQ is marked as busy). #SVOOEXC invokes #RMAEHS. Since the SRM involved is a CRE, #RMAEHS invokes #RMCREAC to process the exception as in the preceding description. In this case, #RMCREAC clears busy on the ACQ and returns control to #RMAEHS that in turn performs an SVX, causing the interrupted receive or dequeue message operation to be resumed and completed. Since the ACQ was marked as busy, other tasks executing a receive message to the ACQ can encounter busy exceptions. In this case, CREs can still be obtained by SVL instructions because these instructions do not observe a busy status on the ACQ.

When the ACQ contains 10 or more CREs and a TDE is being freed, #RMCMBTC is invoked to attempt to free some of the CREs. #RMCMBTC attempts to find four CREs on the same page and return them to storage management by invoking #SMDALCP.

TDE Time-Out

The task interval timer is used to perform time slicing. When the TDE time-out occurs, an exception is generated. #SVOOEXC invokes #RMTSACQ which invokes #RMRPLTS to restore the time slice value.

TDE Availability

The available TDE queue is used to hold the TDEs that are available to VMC tasks. If additional TDEs are required, the queue is replenished from real storage. If the TDE queue contains 10 or more TDEs, the unused TDEs are purged from the queue and returned to real storage.

If a program requires a TDE, the program attempts to obtain a TDE from the available queue. If the queue (RMFRETDE in module #RTTASKS) contains at least one TDE, the first TDE is dequeued and given to the caller. If the available TDE queue is empty, the caller obtains a page from real storage by calling the allocate page frame storage management module (#SMALCPF). The page obtained is split; half is given to the caller in the form of a TDE, and the other half is placed on the available TDE queue. The elements on the available TDE queue are in the sequence according to the address of each element.

When a TDE is no longer required, it is returned by enqueueing the TDE (by its address) onto the available TDE queue. The combined TDE/CRE resource management module (#RMCMBTC) is invoked to purge extra CREs from the available CRE queue and any extra TDEs from the available TDE queue.

The TDE for a process is part of the process control space (PCS) that is created by a Create Process Control Space instruction. The process TDE is not managed as in the preceding description, although all in-use TDEs for VMC tasks or processes are chained in a single list with forward pointers. When a process TDE is no longer needed, #RMCMBTC is invoked to purge any extra CREs from the available CRE queue.

Object Serialization

The object serialization routines support the System/38 locking instructions, implicit locking, object lock enforcement, and the seize/release (#CFSZREL) function used within VMC. A lock or seize is a record stored in a special table called the hold record area. The areas used are shown in Figure 38-1.

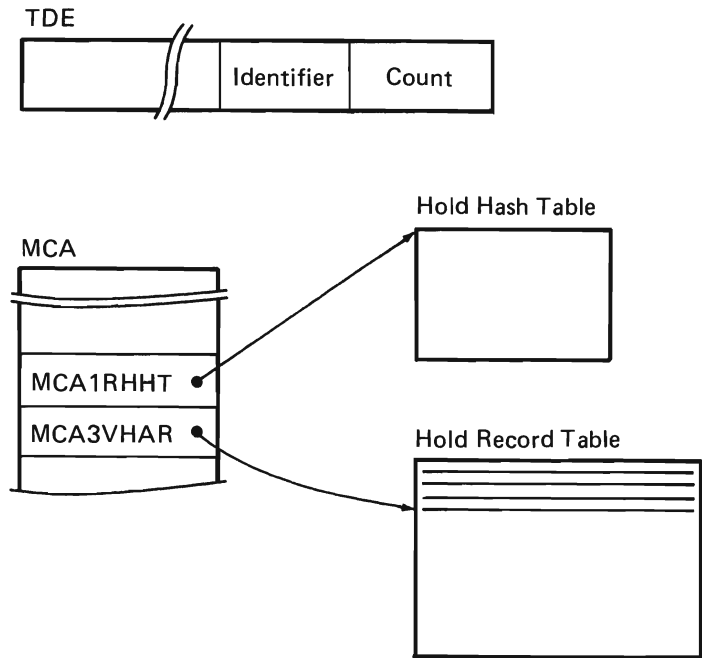


Figure 38-1. Areas Used in Locking or Seizing

An implicit lock is a lock on an object that has been obtained implicitly by a VMC routine supporting an instruction other than a lock instruction. Such locks are normally obtained to ensure that an object is not destroyed. Implicit locks are supported by horizontal interfaces to the routines supporting locking instructions. Lock enforcement refers to the checking done by any VMC routine supporting a System/38 instruction on an object to verify that a lock (implicit or explicit) on the object does not prohibit the instruction. For example, a lock on an object held by process A can prevent process B from operating directly on the object. Lock enforcement is accomplished by checking that no hold conflicting with the desired operation exists in the hold record table. Optionally, lock state checking can be performed on behalf of another task or process. Finally, seize/release is an internal VMC function used to serialize access to an object or a VMC data area during the execution of a single machine instruction. Seize records are stored in the same hold record area as lock records.

The following paragraphs provide a simplified description of these functions. Process A issues a lock against object O. A record is built describing the lock in the hold table. Process B issues a lock on O. VMC searches the hold table, finds the lock record on O, and, if the lock held by A conflicts with the lock requested by B, does one of the following:

- Places B in wait.
- Issues an exception to B.
- Signals an event to B when the lock is obtained while B continues execution.

In the first case, when A issues an unlock on O, VMC restarts B. In the second case, an exception is signaled to B when B attempts to lock O. In the last case, when A issues an unlock on O, VMC signals an object-locked event to B after the lock is obtained for B. Next, suppose B, instead of issuing a lock on O, has issued a destroy. VMC checks whether conflicting locks were held on O (in the case of a destroy, any lock by another process conflicts) and issues an invalid lock state exception to B. This also happens if A had, instead of locking O, issued some other System/38 instruction that caused an implicit lock to be obtained on O.

If process A has to perform a sequence of System/38 instructions that require O to remain unchanged and protected from destruction, A must issue a lock on O to ensure that the instruction sequence can be executed. If A does not issue a lock and executes a System/38 instruction (for example, a materialize instruction) against O, the VMC routine supporting the materialize function issues a seize against O causing a record to be built in the hold table. Again process B issues a destroy. The lock enforcement check is successful since there are no locks on O. However, B is unable to seize O and must wait until the materialize operation of A completes and the object is released.

All serialization-related functions use a group of internal microprogramming instructions called hold/free instructions. For the purposes of this discussion, only the hold and free instructions are considered here. Lock, seize, and lock enforcement are performed by a hold, and unlock and release are performed by a free.

The hold/free instructions use the areas shown in Figure 38-1. The TDE contains a unique 2-byte identification field (TDEID) and a 2-byte count of hold records chained from this TDE (TDEHLDCT). The identification field is maintained by GETTDE while a create task or initiate process initializes TDEHLDCT to 0. Within the machine communications area (MCA), a field (MCA1RHHT) points to the hold hash table (which is a VMC nucleus module) and MCA3VHAR points to the first available hold record. MCA1RHHT is set by the link-loader. MCA3VHAR is initialized by #RMINIT when the hold record area is allocated and built by #RMINIT at initial program load (IPL) time.

A hold request specifies:

- The address of the object involved.
- A byte describing the types of hold states that are to be interpreted as a conflict. (Hold state refers both to lock states and seizes.)
- A byte specifying the types of hold states to be obtained if there are no conflicts.

All addresses that have the same hash index are placed on the same hold record chain, and all holds for identical addresses are placed on a secondary chain. All activity on the secondary chain is done by the VMC. Searching the chain and then adding or removing hold records is started when the HMC signals a second chain exception.

If a hold can be added or removed from the secondary chain, then the VMC exception handler completes as if the grant instruction or the free instruction is executed successfully. If any unusual condition is found while searching the secondary chain, the exception is changed to a chain-conflict or end-of-chain exception.

The hold instruction performs a hash on the object address to locate the start of a (possibly empty) chain of hold records within the hold record area. The chain is searched for records specifying the same object address. If such a record is found and the TDE identification is the same, the record is ignored and the search continues. If the TDE identification is different and the lock state specified in the found record does not conflict with the requirement specified by the hold instruction, the search continues. If the search reaches the end of the chain, a new hold record is built. If a conflict was found, a machine exception is raised. An exception handler (#RMGFEX) then takes the proper action by placing the requestor in a wait, and either raising an exception or returning control to the requester with an indication of the conflict.

If an action must be taken when the hold is freed (using a release or unlock), an indicator (the monitor flag) is set in the hold record by the exception handler. A Free Hold Record or Free Hold Record First instruction causes the chain to be searched as for hold. When a record containing the specified object address and lock state for the issuing TDE is found and the monitor flag is on, an exception is raised. The exception handler (#RMGFEX) then takes the appropriate action (for example, dispatching a waiter). If the flag is off (the normal case), the free instruction simply returns the hold record to a chain of free records (available for subsequent holds).

Seize/Release

Within VMC, seize/release is used to control object access. It can be used for machine interface object or any VMC internal object. Object serialization due to seize is not visible at the machine interface except as a delay in the execution of an instruction. When there is an attempt to seize an already seized object, the task or process waits until the object becomes available. If there is more than one waiter for the object, they are all dispatched and allowed to compete to seize the object.

Two types of seize are available: shared or exclusive. Any number of TDEs can seize an object for shared use. If an object is seized for exclusive use, no other TDE can seize the object.

Module #CFSZ is invoked to perform the seize function; module #CFRLS is invoked to perform the release function. These modules use the register save area in the seize/release parameter area. When a module is invoked, R1 points directly to the seize/release parameter area.

Within VMC, deadlock is avoided by using a seize hierarchy.

Objects are seized in the order specified in the seize ordering table (#CFSZORD). A space object has the lowest number and is the first object seized. The machine context has the highest number and is the last object seized. A value of 00 indicates a type of object that is not seized.

Seize in general avoids deadlock by following a specific object order. This order is defined only by the object type. Deadlock occurs when two processes each hold an object and the processes are waiting for the object held by the other process. As long as the general type order is followed, seize avoids deadlock within one seize request by not holding any object listed in the request if it is necessary to wait for another object. If the first object in the request is obtained and the second is not available, the first is released before waiting for the second. When the second object becomes available, the entire seize request is restarted from the beginning. Ordering is not required for objects listed for one seize with one restriction. The restriction is that an object cannot be seized exclusive if previously seized shared because this can result in a deadlock wait.

At initial program load (IPL) time, #RMINIT initializes the following data areas in the seize-wait data area (#RMSZDX):

- A queue of available message
- Four areas, each area containing:
 - A counter used as a gate
 - A queue used to record the TDEs waiting for an object

The data associated with one execution of seize or release is contained in the seize/release input area (ZZSRP). This area contains the number of objects to seize, the address of each object, and the type of seize for each object.

When an object is seized, a Grant Hold Record or Grant Hold Record First instruction is executed. If there is no conflict, the internal microprogramming instruction records the type of hold and the seize is completed.

If there is a conflict, the TDE must wait for the current conflicting seize to be released. Routine #RMSLHCS within #RMGFEX is given control from the exception handler (#SV00EXC). Two bits of the object address are used to select one of the four seize wait data areas. Access to the area is serialized by a send/receive counter and a message recording the object address, and the waiting TDE is placed on the object queue. The conflicting hold record is then marked monitored and the busy bit is turned off in the object chain. Then #RMSLHCS determines if other objects were already seized by the current use of seize. If so, they are released to avoid deadlock. Finally, a Receive Count instruction is performed to the wait count in the TDE to actually put the task or process into a wait. When the TDE receives the count (due to a release), the registers in the exception CRE and instruction address register (IAR) are reset to again start seizing with the first object requested in the current use of seize. Then #RMSLHCS does a Supervisor Exit instruction to retry the seize.

When an object is released, a Free Hold Record or Free Hold Record First instruction is executed. If the hold record is not monitored, the hold is removed from the object chain and the release is completed.

When a hold is monitored and exception is raised, routine #RMSLMHS within #RMGFEX is given control by the exception handler. This routine uses the object address to select the seize data area to access. The area is serialized by doing a receive count, and the messages listing the waiters are removed from the object queue. A count is then sent to the wait counter for each waiter. Finally, the monitored and busy flags are turned off and a Supervisor Exit instruction results in retry of the free instruction.

Lock Enforcement

Object locks on system objects are enforced by VMC. This means that before the VMC accesses a system object, the lock state of the object is checked. If a lock prohibits the attempted use of the object, an exception is signaled.

The lock state of an object is checked by using Grant Hold Record and Grant Hold Record First instructions with test byte bits set corresponding to the lock states which would prohibit the operation. The hold request bits are all zero so a successful instruction will not actually add a new hold record to the object chain.

It is possible to request checking on more than one object. The number of objects, the address of each object and the lock states to check for each object are stored in the seize/release parameter structure. Lock state checking proceeds one object at a time until all objects have been checked or a conflicting lock state is found.

If there is a conflicting lock, an exception is raised and routine #RMSHCL within #RMGFEX is given control by the exception handler.

#RMSLHCL clears the busy bit in the object hold record chain, sets a return code in the structure seize/release parameter, advances the IAR in the exception CRE past the Grant Hold Record or Grant Hold Record First instruction, and returns to the VMC routine that was checking lock states.

Because seize and lock use the Grant Hold Record and Grant Hold Record First instructions, it is possible to combine seize and check lock state in one operation. When this is done, the test and hold bits for seize and lock checking are simply combined. If there is a conflict due to lock state, there is no change in the exception processing. If the conflict is due to the requested seize only, the exception processing proceeds as if the operation was a seize only.

Lock Support

The following list shows the System/38 locking instructions and the various horizontal VMC interfaces for performing lock-related functions:

System/38 Instructions

Lock Object	Lock one or more system objects to a process.
Unlock Object	Unlock one or more system objects from a process, cancel a specific object lock wait, or cancel all asynchronous lock waits for a process.
Lock Space Address	Lock one address in a space to a process.
Unlock Space Address	Unlock one address in a space from a process.
Transfer Object Lock	Transfer locks held by a process on one or more system objects to another process.
Materialize Object Locks	Materialize a list of all locks held by processes on an object.
Materialize Process Locks	Materialize a list of all locks held by a process.
Materialize Selected Locks	Materialize a list of locks held by the current process on the specified object or space address.
Materialize Allocated Object Locks	Materialize the current allocated locks on a designated object.

Horizontal Interfaces

Lock	Implicitly lock an object to a process or task.
Uniock	Unlock an implicit lock.
Transfer Lock	Transfer an implicit lock.
Cancel Synchronous Lock Wait	Cancel a synchronously waiting request if the invocation is terminated.
Unlock for Terminated Process	Unlock all objects held by a process when the process is terminated.
Unlock for Destroyed Object	Unlock all locks on a given object when the object is destroyed.

Lock Object: When the System/38 Lock Object instruction is executed, #RMLK is invoked by the SVL router. #RMLK first copies and validates all the input data. Then, the input system pointers are validated and resolved. If an error is found, an exception is signaled.

Actual locking is done using the Grant Hold Record and Grant Hold Record First instructions. If there is no conflict, execution of the Grant Hold Record or Grant Hold Record First instruction completes the lock operation for one lock. A loop is set up to repeat this for each lock requested.

If there is a conflict, a machine exception is generated and #RMSLHC receives control from the exception handler (#SV00EXC). #RMSLHC first unlocks any locks already obtained for this Lock Object instruction. Then the option specified by the program is examined. If an immediate exception is wanted, the busy bit in the object hold chain is cleared and machine interface exception is signaled.

If the program requests a synchronous wait for the lock, the conflicting hold record is marked as monitored, a message defining the process and object waited for is placed on a queue, and the busy bit is cleared. Then the common functions #CFRWTO is used to wait for a message indicating the conflicting lock is unlocked or the wait time has expired. If the conflicting lock is unlocked, the process again tries to obtain the requested locks starting with the first lock listed for this Lock Object instruction. If a wait time is specified and the time expires, an exception is signaled.

If the program requests an asynchronous wait, the data defining the locks to obtain is first copied to machine-wide storage (MWS). A message is then placed on a queue defining the process and object waited for. Next, if a wait time is specified, #CFWTO is invoked to make a timer request and send a message if the wait time expires. The IAR is advanced, and an SVX operation is performed to return as if all locks were granted.

If the object is unlocked, the timer request (if any) is canceled and the originally requesting process is interrupted to retry locking. The retry is executed by #RMLKAR. This routine finds the list of locks to obtain in the MWS area which was obtained by the original Lock Object instruction.

If the time limit is specified and that time limit expires for the task receiving the time-out message, the resource management service task #RMSVTSK invokes #RMLKTO to remove the message that indicates the process is waiting, and then signals a time-out event.

A horizontal interface to the lock function is provided so that VMC routines can implicitly lock objects. The name of this routine is #RMHLK.

Unlock: When the Unlock Object instruction is executed, the SVL router passes control to #RMUNLK. #RMUNLK first copies and validates the input data. Then the input system pointers are checked. An exception is signaled if an error is detected. The unlocking is actually done by setting up a loop to execute the Free Hold Record and Free Hold Record First instructions for each lock specified.

If the process does not have such a lock to unlock, an internal microprogram exception is signaled and routine #RMSLNHL within #RMGFEX will receive control. The exception handler will set a return code in the input parameters and continue as if the operation was successful. When all requests have been processed, there is a check for errors. If one or more locks were not unlocked, an exception is signaled indicating which locks were not unlocked.

If another process is waiting to obtain a lock that is being unlocked, the hold record was marked as monitored when the other process encountered the conflict. When there is an attempt to free a monitored hold, there is an internal microprogram exception and #RMSLMHL receives control. This routine first sets the hold byte to zero and frees the hold so the hold chain will not be held busy longer than necessary. Then the list of waiters is examined. If a wait is synchronous, a message is sent to allow the process to resume and retry the lock operations. If a lock wait is asynchronous, the waiting process is interrupted so it will in effect reexecute the Lock Object instruction.

A horizontal interface, #RMHUNLK, is also provided so VMC routines can unlock implicit locks.

Cancel a Synchronously Waiting Lock Request: When an invocation containing a synchronous lock wait is terminated, the lock request is canceled. #RMSLWT is invoked when an invocation is terminated which contained a waiting lock request. This routine removes the request from the waiting message SRQ and cancels the time-out request. No event or exception is signaled to machine interface.

Unlock of Destroyed Object: When an object is destroyed, the object may be locked. If a lock exists, it must be unlocked and waiters, if any, must be dispatched.

Routine #RMDOULK is invoked during the final destruction of an object by the process destroying the object from #CFDESTO. It sets the hold record chain for the affected object address busy, copies any lock records for the object to local storage, and clears the busy flag. Next, a free is issued for each lock held on the object, based on the list of copied lock requests. (Thus, the copying serves to form a series of unlock templates.) If any of the locks on the object have caused other processes to wait, normal unlock logic in #RMSLMH causes the waiters to be dispatched or interrupted. (The object is now destroyed, however, and any reference to it results in a destroyed object exception.)

Unlock Objects Locked to a Terminated Process: This routine, #RMDPULK, is a horizontal interface from terminate process. A process must unlock any locks it holds before it can be terminated. This routine executes under the to-be-terminated process.

Because the hold records are not chained by TDE or process identification, the entire table of hold records must be searched to locate the locks held by a process. Each time a lock is located, a free instruction is executed to unlock the lock and re-dispatch or interrupt any waiters. The search for hold records is not done unless it is possible for the to-be-destroyed process to hold a lock.

Transfer Lock Object: When Transfer Object Lock instruction is issued, the SVL router passes control to #RMMMCLK. This routine copies the input data, checks the input for valid options, and validates and resolves the object pointers. If an error is detected, an exception is signaled.

A loop to process the requests is established. In this loop, the hold chain for an object is set busy and the entire chain is searched. This search is to locate the specific lock to transfer and all locks on the object held by the process. If the transfer does not violate the lock granting rules, the current owner of the hold record is modified and the address of the hold record is saved. The busy flag for the object chain is then reset.

The list of hold record addresses for all locks that meet the test for a valid transfer is sent to the receiving process. #RMINIPI interrupts the receiving process to receive the list.

If an error condition is detected before the locks are transferred, the exception data is saved; after transferring all locks which can be transferred, an exception is signaled.

When the receiving process is interrupted, it invokes routine #RMRCVLK. #RMRCVLK removes the list of locks from the process interrupt queue. The receiving process places its TDE identification in the indicated hold records. The receiving process completes the transfer so that event handling, lock transfers, and process destruction can be properly serialized.

The receiving process could be in a lock-wait waiting for one of the locks which was just transferred to it. So that the receiving process does not resume a wait state after receiving the locks, the receiving process checks the list of lock waiters to see if it was waiting for the lock just transferred. If this is the case, the receiving process completes transferring the locks and sends a message to resume the suspended lock operation.

Horizontal Interface to Transfer Lock: When a process is created, the Initiate Process instruction can optionally specify the locks to be transferred to the created process. In this case, #RMHMXLK is invoked from a process creation routine. (#RMHMXLK is an entry point in the main transfer lock routine.)

Also, when a process is created, the process must be given a lock on the user profile it runs under. This is done by #RMHXLK. This routine contains limited checking for this special case and transfers only one lock.

When a user profile is changed by the Modify Process Attributes instruction, #RMHIXLK is invoked to transfer the lock. This is also an entry point in the main transfer lock routine. The transfer logic and checking is the same but the input is not a template.

Materialize Object Locks: The SVL router passes control to #RMMTOLK when the Materialize Object Locks instruction is executed. #RMMTOLK first verifies the input and locates the object.

The Set Chain Busy internal microprogramming instruction is used to serialize access to the list of locks for this object. The chain of hold records is searched for locks on the selected object. The desired lock entries are copied to a work area and the chain is made not busy.

The data is moved to the output area specified by the System/38 instruction. If the output area is not large enough to contain the data, the excess data is truncated.

Finally, the list of lock waiters is examined. Any waiters for this object are copied. Then the data is moved to the output area, again truncating excessive data if the size of the output area is exceeded.

Materialize Process Locks: #RMMTPLK is given control by the SVL router when the Materialize Process Locks instruction is executed. #RMMTPLK tries to locate the specified process. An exception is signaled if the process cannot be found.

Materialize Selected Locks: #RMMTSLK is given control by the SVL router when the Materialize Selected Locks instruction is executed. #RMMTSLK is designed to give a list of the locks held by the current process on one specific object or space address.

#RMMTSLK is basically a short, faster form of materialize object locks. However, no lock waits are materialized and only locks by the current process are materialized.

The output data is moved to the user's space until all data is materialized or the space provided is filled.

The table of hold records and the current size of the table are located and the table is searched for any locks held by the specified process. The table is searched sequentially because there is no chaining based on TDE identification. When locks held by the process are located, the lock data is copied to a work area.

Note: The table may have been extended at the same time it was being searched. After completing the search, the size must be checked to see if additional records have been added.

The lock data is moved to the output areas specified by the System/38 instruction. If the output area is not large enough to contain the data, the excess data is truncated.

Finally, the list of lock waiters is examined. Any waits outstanding from this process are copied. Then the data is moved to the output area again, truncating excessive data if the size of the output area is exceeded.

Initialization

When an object is locked successfully, a hold record is created. When there is a conflict, the exception handler records the waiter data in the lock wait data area (#RMLKDX) if a wait option was selected. This area is initialized at IPL by #RMINIT. This area consists of a queue of currently available messages and a send/receive counter used to serialize access to the available message queue and the eight lock wait areas. Each wait area consists of a send/receive count (SRC), a queue to hold a message for each lock being waited for, and a queue where processes waiting synchronously for a lock do a receive message to wait for the lock to become available. When it is necessary to wait for a lock, the wait area is selected by using 3 bits of the waited-for object address.

After copying from the program template, the data for one execution of lock, unlock, or transfer lock is found in the lock/unlock input area ZZLKI. This area contains some fields not needed for every instruction, such as the wait time and a pointer to the parameters used to seize the objects. Following these fields is a count of the number of objects, the object addresses, and the lock states.

Timer Services

Timer service routines provide support for the System/38 instruction that includes a wait time limit. These routines are used exclusively by other VMC components that support the System/38 instruction set.

Timer service functions use the internal microprogram clock comparator and the time-of-day clock. When the clock comparator matches the time-of-day clock, HMC sends a count to the send/receive counter at the address contained in the machine communications area (at MCA2VSCC). The address of the clock comparator is externally known and is set by the link/loader.

A task created at IPL by #RMINIT waits on the send/receive counter associated with the clock comparator. When a count is received, a message is dequeued from a queue of timer requests and this task performs the requested service. The task again does a receive count to the clock comparator send/receive counter. The program being executed by this task is #RMCCINT.

All data areas used by the clock comparator are contained in the clock comparator data area (#RMCCDX). This area contains the send/receive counter that is set by the clock comparator, a queue used for outstanding timer requests, a queue used for available (unused) requests, a send/receive counter used as a gate to serialize access to this area, and some initially available request messages.

At IPL time, #RMINIT uses the clear-page-and-pin function to obtain 512 pinned bytes of 0's for this area. The pin is used because a send count to the clock comparator counter cannot page fault. Clear-page is used so that it is not necessary to read a VMC page which is then cleared. #RMINIT also enqueues the initially available messages to the available message queue.

When a request is made for a timer service, a message is removed from the available queue, data is stored defining the user request, and the message is placed on the queue of outstanding requests.

Timer services support the following System/38 instructions:

- Modify Machine Attributes (Time-of-Day)
- Materialize Machine Attributes (Time-of-Day)

In addition to the preceding instruction support, timer services provides the following support:

- Wait time-out
- Receive with time-out
- Wait time-out for event
- Wait time-out request cancel
- Extend time-out wait request queue

Modify Machine Attributes (Time-Of-Day)

#RMMDTOD is invoked to modify the time-of-day. This module sets the time-of-day value to the specified value and updates all outstanding requests for timer services for a time interval so that the interval is terminated at the correct time.

Materialize Machine Attributes (Time-Of-Day)

#RMMTTOD is invoked to materialize the current time-of-day. This module stores the time-of-day value into the specified user area.

Wait Time-Out

#CFWTO is used to request that a count or message be sent after a time interval. Optionally, this action can be repeated indefinitely. #CFWTO returns to the caller as soon as the request is placed on the queue. Parameters passed to this routine are contained in area ZZWTO.

Receive with Time-Out

#CFRWTO is used to execute a receive message with a limit on the time that the process will wait for the message. #CFRWTO does not return to the caller until a message is received or the time limit is exceeded. The input to this function is contained in area ZZRWT. This function can be used only by machine interface processes.

Wait Time-Out for Event

#RMWTOE is invoked to request interrupting a process for an event after time interval. #RMWTOE returns to the caller as soon as the request is placed on the queue. The input to this function is contained in area ZZWTO.

Wait Time-Out Cancel

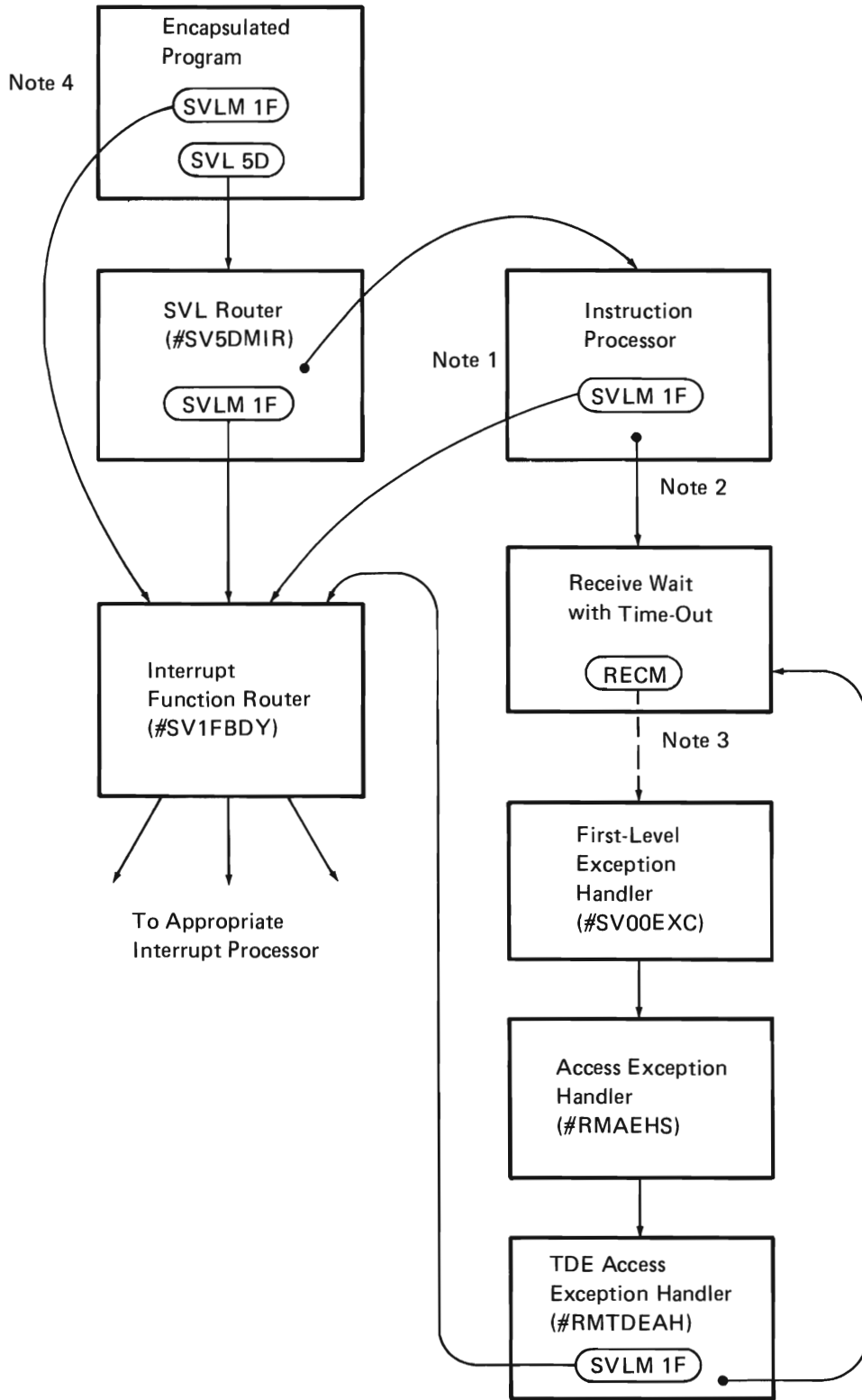
#CFWTOC is invoked to cancel either a specific request or all requests for a task or process. #CFWTOC returns to the caller when the request is canceled. The input to this function is in area ZZWTC.

Extend Wait-Request Queue

#RMGTREQ is invoked if another request message is needed and none are currently available. This routine obtains a page of machine-wide storage, formats the page into SRMs, places the new SRMs on the available queue, and returns a new SRM to the caller.

Process Interruption

Process interruption provides the functions that interrupt the execution of a process in an orderly fashion to perform or schedule other processing that usually is not related to the activity of the current process. Figure 38-2 shows an overview of process interruption.



Notes:

1. This is the path used for long-running VMC routines.
2. This is the path used for an instruction wait.
3. SV00EXC is invoked as a result of a TDE access exception.
4. This linkage occurs at an instruction boundary.

Figure 38-2. Servicing Pending Interrupts

Process interruption can be initiated at any time, but the processing of the interrupt function under the target process (interrupted process) always occurs at System/38 instruction boundaries. System/38 instruction boundaries are defined as follows:

- In translator generated code: A backward branch that could create a loop
- In the SVL router (#SV5DMIR): Before and after the System/38 instruction processor module is invoked
- In System/38 instruction wait: When a System/38 instruction process is in the wait state as a result of executing one of the following instructions:
 - Lock Object
 - Dequeue (with wait option)
 - Wait on Event
 - Set Cursor (with lock-entry-with-wait option)
 - Suspend Process
- In long-running instruction processors

Process interruption is accomplished by using the Supervisor Link Monitored instruction. The Supervisor Link Monitored instructions are inserted at System/38 instruction boundaries. The Supervisor Link Monitored instruction is executed as a no-op unless enabled by a bit in the TDE. When enabled, the Supervisor Link Monitored instruction executes as a Supervisor Link Short instruction. In translator generated code, the translator inserts the Supervisor Link Monitored instruction before a backward branch and a branch where the target is not known at the time of translation. The Supervisor Link Monitored instructions are inserted into the SVL router before and after invocations to VMC modules that process the System/38 instructions. The Supervisor Link Monitored instructions are inserted into long-running System/38 instruction processors at points where processing can be interrupted.

A process can be interrupted to perform the following functions:

- Time slice end processing (signal event and MPL control)
- Receive transferred locks
- Asynchronous lock request lock retry
- Suspend process
- Resume process
- Modify process attributes
- Terminate machine processing
- Terminate process
- Wait time-out processing
- Schedule event

A process in an instruction wait can be interrupted from a wait state to perform the preceding functions except when the process is suspended (in suspend-wait). If suspended, the process can be interrupted only by resume process, terminate machine processing, or terminate process functions.

A process is usually interrupted by another machine interface process or a VMC task, but a process can also interrupt itself. For example, when an asynchronous exception such as time slice end is presented, some of the processing for the exception can be delayed until the current process reaches the next System/38 instruction boundary.

Only machine interface processes can be interrupted; VMC tasks cannot be interrupted.

Initiate Process Interrupt

#RMINIPI performs the initiate process interrupt function. This routine sets the supervisor link monitor (SVLM) enable bit in the TDE of the target process, optionally enqueues a message to the process-interrupt-data SRQ that is in the resident portion of the process control block (PCB) (PCBRES) of the target process, and sets a bit in the function code (also in PCBRES) to indicate which interrupt function to perform. If the target process is not in instruction wait, #RMINIPI returns control to the caller; the pending interrupts are then serviced at the next System/38 instruction boundary of the target process.

If the target process is in an instruction wait, #RMINIPI enables TDE exceptions, dequeues the process from the wait SRQ and enqueues it to the prime task dispatching queue (TDQ), and then dispatches the process. #RMINIPI then returns control to the caller.

Interrupt Function Router

The interrupt function router (#SV1FBDY) receives control from an enabled Supervisor Link Monitored instruction. The function code in PCBRES is saved in the CRE obtained for the Supervisor Link Monitored instruction. The function code is cleared and further processing is based on function code saved in the CRE of the Supervisor Link Monitored instruction. #SV1FBDY invokes the appropriate interrupt function handler(s) and clears the corresponding bit(s) in the CRE of the Supervisor Link Monitored instruction. #SV1FBDY returns control (SVX) to the caller when all pending interrupt functions are handled. The paths to #SV1FBDY are shown in Figure 38-2.

Receive Message with Wait-Time-Out

The Receive Message instruction with wait-time-out function (#CFRWTO) is invoked by the System/38 instruction processors that can place a process in an instruction wait. #CFRWTO issues the Receive Message instruction and provides the wait with time-out function. #CFRWTO establishes a real-time time-out by using the timer service function based on a specified time-out interval value. #CFRWTO sets the wait bits to indicate the intended instruction wait and enables the TDE access exception before issuing a Receive Message instruction to the designated queue. If no message is received, a TDE access exception is presented and control given (through #SVOOEXC and #RMAEHS) to the TDE access exception handler, #RMTDEAH. #RMTDEAH eventually returns (SVX) to reexecute the Receive Message instruction in #CFRWTO and put the process in a receive wait (instruction wait). If the intended message is received, the TDE exception is disabled, the wait bits are reset, and the wait time-out request is canceled (by a call to #CFWTOC), and the received message returned to the invoking module. If wait time interval expired before the Receive Message instruction is satisfied, #CFRWTO resets the wait bits, enables the TDE exception, sets the time-out return code, and returns control to the calling module.

TDE Access Exception Handler

When a TDE access exception is presented during an unsuccessful Receive Message instruction, the exception is presented through the Supervisor Link Short instruction that passes control to the first-level exception handler (#SVOOEXC). #SVOOEXC invokes #RMAEHS (TDE, SRM and SRQ access exception handler); #RMAEHS invokes #RMTDEAH to handle the exception. #RMTDEAH disables TDE exceptions and checks the interrupt enabled bit. If the interrupt bit is on (interrupt enabled), #RMTDEAH frees the SRQ, issues the Supervisor Link Monitored instruction, and passes control to #SV1FBDY to service the interrupt.

Interrupt functions are serviced before a process enters instruction wait state. This is also the execution path for process that is interrupted from an instruction wait. When #RMTDEAH regains control from #SV1FBDY, #RMTDEAH performs an SVX that causes #CFRWTO to reexecute the Receive Message instruction. The process remains in the instruction wait state if the Receive Message instruction is unsuccessful.

Receive-Wait Time-Out Handler

When an instruction wait expires, the real-time clock comparator service module, #RMCCINT, that runs under a VMC task receives control. #RMCCINT interrupts the time-out process out of an instruction wait to perform wait time-out processing in module #RMMIWTO. A process is interrupted out of an instruction wait using the TDE access exception. The TDE-access-exception-CRE contains the status of the #CFRWTO code when the Receive Message instruction was executed. Therefore, #RMMIWTO advances the IAR in the TDE-access-exception-CRE to the time-out return entry in #CFRWTO. Then, #RMMIWTO returns control to #SV1FBDY that returns control (SVX) to #RMTDEAH. When #RMTDEAH issues the Supervisor Exit instruction, the status of #CFRWTO is restored from the TDE-access-exception-CRE. The Receive Message instruction will not be reexecuted; instead, a return code that indicates time-out is set. When #CFRWTO returns to the caller, the calling module checks the return code and signals a wait time-out exception at the machine interface. #RMMIWTO also returns the time-out message to timer services available message queue.

Exception Back Out

If an exception causes the invocation of #CFRWTO to be terminated, #RMMIETO is called to cancel the time-out request or to back out the time-out message if time-out has already occurred, and #RMPICRE is called to restore the not yet serviced interrupt function in the CRE(s) obtained for the SVLM(s). Both of these modules can be called by return from exception module #EXRTEXP, or by exception generator, #CFEXGEN. #RMMIETO invokes #CFWTOC to cancel wait time-out request. Then it calls #RMMIWTO to back out and return the time-out message if the wait has already expired.

#RMPICRE restores the unserved interrupt functions from CRE(s) of the SVLM to the function code field in PCBRES and clears the save area in the CRE(s) before exception management makes the CRE from in-use state to available state. The function of the not yet serviced function is restored as process interruption is asynchronous to the executed sequence of the interrupted process; the termination of invocations does not mean termination of pending interrupts.

Serialization of Process Interruption

The process that initiates process interruption (the interrupting process) against another process (the target process) ensures that the target process is not terminated or in its final termination phase. The interrupting process uses the Compare and Swap Halfword instruction to determine the status of the target process. If the target process is being terminated, a nonzero return code is returned and the interrupting process is not allowed to interrupt the target process. If the target process is not being terminated, a halfword in the resident portion of the PCB is incremented by the Compare and Swap Halfword instruction to prevent the target process from terminating.

When the interrupting process completes processing of the interrupt, it uses the Add Logical Halfword Immediate instruction to decrement the halfword in the PCB.

When the value of the halfword in the PCB is greater than 0, a process cannot begin the final termination phase. A process uses the Compare and Swap Halfword instruction to check the value in the halfword. If the halfword is 0, the value is set to a -1 (minus one) value (this value prevents any further interruptions); if the halfword is not 0, then the process waits on an SRC in the resident PCB. A process waiting to enter the final termination phase is released when the current interrupting process(es) complete the interruption(s). If a process attempts to interrupt itself and the halfword in the PCB has a -1 value, a critical bit in the TDE is set. In this case, a terminate immediate internal microprogram instruction is executed to terminate machine processing.

Multiprogramming Level Support

Resource management MPL support consists of VMC routines that support the MPL definitions and rules. Following are the definitions as they relate to MPL support:

- **MPL:** Multiprogramming level is the number of processes currently executing. A process is active or in the current MPL if it is executing.
- **Machine-wide MPL (current):** The number of user processes currently executing in the machine.
- **Machine-wide MPL (maximum):** A user specified value for the maximum number of processes which can execute concurrently in the machine.
- **MPL class:** A logical set of processes. A process is assigned an MPL class at process initiation by the user. The MPL class can be altered by a Modify Process Attributes instruction.
- **Class MPL (current):** The number of user processes currently executing in a class.
- **Class MPL (maximum):** A user-specified value for the maximum number of processes that can execute concurrently in a specified MPL class.
- **System/38 instruction wait:** A process is in a wait either as a result of a Suspend Process instruction that has been issued or an unsatisfied Lock Object, Wait on Event, Dequeue, or Set Cursor (that causes an implicit lock for update or delete) that has been issued. When a process enters a wait, the process is not considered as part of the current machine-wide or class MPL.
- **Ineligible wait:** A process is in an ineligible wait when it is neither executing (active) nor in a System/38 instruction wait. It is placed in this state as a result of applying the MPL rules.
- **Time slice:** A user-specified amount of processor time a process can use before being subject to MPL rules.

System/38 instruction set process is eligible to execute only if the machine-wide MPL maximum and class MPL maximum (for the class to which it is assigned) permit it to be active; for example, when the count of active processes does not exceed the maximum values. Figure 38-3 shows an overview of the application of the MPL rules. The rules are enforced as follows:

- When a process is initiated or leaves the System/38 instruction wait state (because its wait was satisfied or to perform an interrupt function such as handling an event), it is added to the current MPL only if the machine-wide and class MPL limits permit. Otherwise, it is placed in the ineligible state. (Thus, it does not preempt processes currently in the active state.)
- When a process enters the System/38 instruction wait state, terminates, or the machine-wide or class MPL maximum values are increased, the selection algorithm is dispatched to select and redispach ineligible processes.
- When a process reaches time slice end and there are processes of equal or higher priority that are currently ineligible (but could be made active if the current processes were removed from the MPL), the current process is made ineligible and the selection algorithm is invoked. Otherwise, the current process resumes execution with a full time slice value after processes of equal priority on the prime TDQ.

The selection algorithm proceeds as follows:

The search begins with the ineligible waiting process that has the highest priority. The process is selected if both the following conditions are met:

- The machine-wide current MPL is less than the machine-wide maximum MPL
- The MPL class current MPL is less than the MPL class maximum MPL

The selection procedure continues until one of the following occurs:

- The machine-wide current MPL equals the machine-wide maximum
- Remaining ineligible processes cannot be selected (MPL class current MPL equals the MPL class maximum MPL for these processes)
- There are no processes left in an ineligible wait state

If there is any process left in the ineligible wait state, a test is made to determine if the ineligible threshold exceeded event(s) should be signaled.

During the application of MPL rules, counts are maintained (by machine-wide and MPL class) of the number of processes in each state and the number of transitions between states. These counts can be materialized via the Materialize Resource Management Data instruction.

If machine-wide maximum MPL value is equal to or greater than the sum of all class maximum MPL values, it has no effect. Processes in each MPL class compete among themselves for the current class MPL slots on a priority basis. There is no competition for the current MPL slot among processes belonging to different MPL classes.

If machine-wide maximum MPL value is less than the sum of all class maximum MPL values, all processes compete for the current machine-wide MPL slots on a priority basis.

In both of the preceding cases, an executing process (a process in the current MPL set) cannot be preempted from the current MPL set by another process until the executing process encounters time slice end or enters System/38 instruction wait. That is, priority is used to enter the current MPL set only when there is an open slot in the current MPL set.

The only exception to the MPL rules is the System/38 Dequeue (wait state) instruction with the stay-in-MPL option. When that option is specified, the process remains in the current MPL set even when it enters the wait state. This option allows processes that expect a short wait, such as waiting for a response from high-speed source/sink devices, to not incur the overhead of leaving and entering the current MPL set. Deadlock can occur if a process is holding a current MPL slot and waiting on a queue for a message, while the process that is to send the message is waiting for the current MPL slot held by the first process.

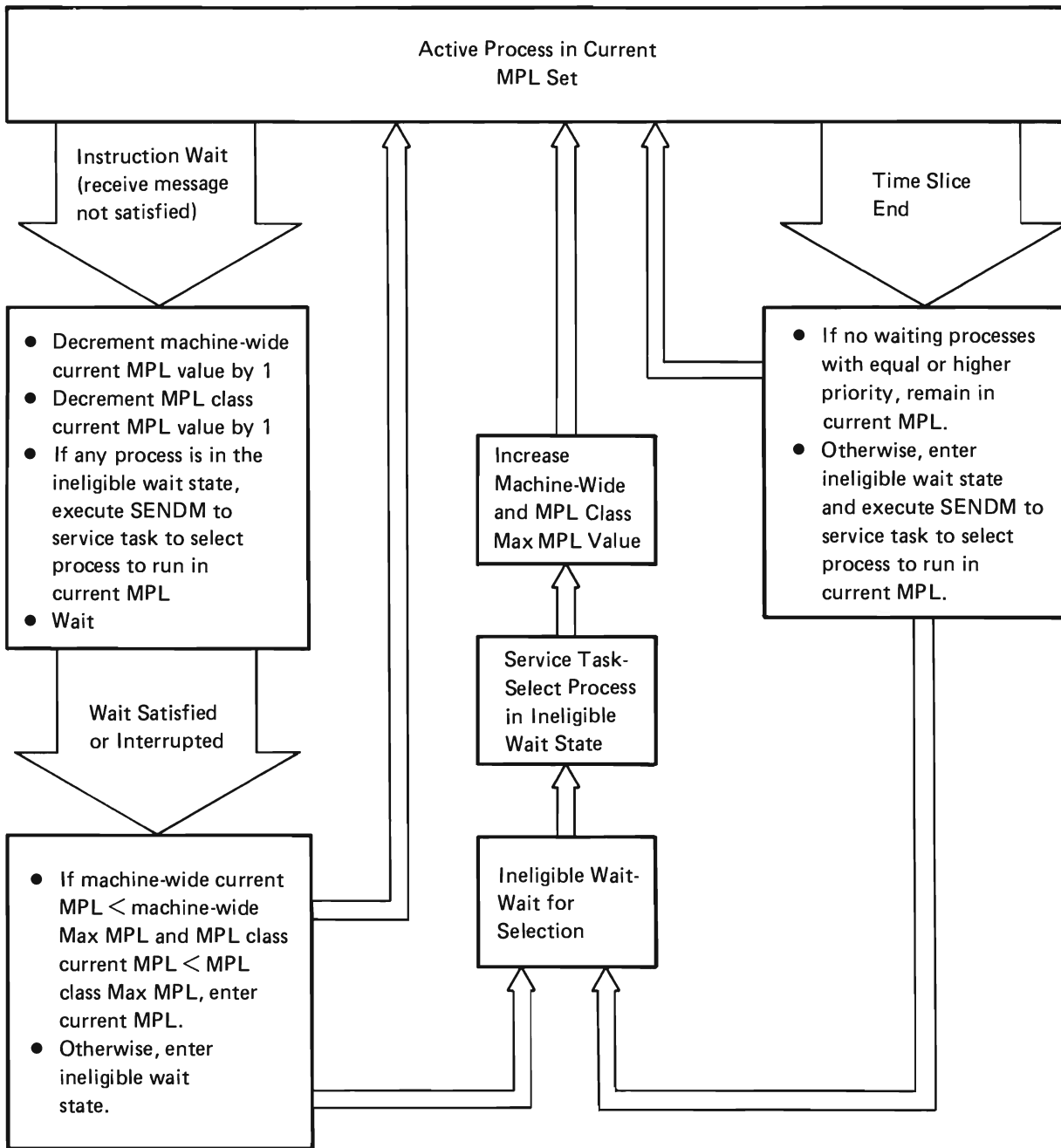


Figure 38-3. Application of MPL Rules

System/38 Instruction Wait

Lock Object, Wait on Event, Dequeue, Set Cursor, and Suspend Process instructions cause the issuing process to enter a System/38 instruction wait state. The instruction processors for these System/38 instructions invoke the receive with wait-time-out function (#CFRWTO). This function establishes a timer mechanism that allows the invoking process to be restarted after a specified time interval if the conditions of the wait are not satisfied within the time interval.

When #CFRWTO receives control, it enables TDE access exceptions and executes a Receive Message instruction to the specified SRQ. This Receive Message instruction causes the invoking process to get a TDE access exception. The TDE access exception is routed through the first-level exception handler (#SV00EXC) and the access exception handler (#RMAEHS) to the TDE access exception handler (#RMTDEAH).

The TDE access exception handler inhibits any additional TDE access exceptions, resets the SRQ to not busy (it was marked busy by the interrupted Receive Message instruction), and invokes #RMRPLTS to restore the time slice if the process access group is to be purged when the System/38 instruction wait state is entered. #RMTDEAH then executes a Supervisor Link Monitored instruction. If the Supervisor Link Monitored instruction is enabled (indicating that an interrupt is pending), the Supervisor Link Monitored instruction is executed as a Supervisor Link Short instruction and control is passed to the interrupt function router (#SV1FBDY). #SV1FBDY processes the interrupt(s) and returns control (SVX) to #RMTDEAH.

#RMTDEAH then invokes #RMMINWT if the current process is in the current MPL and will not remain in the current MPL when the System/38 instruction wait is entered.

#RMMIMWT performs the MPL-related functions, it updates the MPL counts (which are maintained in #RMMPLM), issues a purge (storage management perform paging request, which invokes #SVE8PPR) against the process access group if the appropriate indicators are on, and determines if there are processes on the ineligible queue. If there are processes on this queue, a message (which is preformatted and resides in module #RMMPLM) is sent to the service task queue in #RMMPLM. This causes the service task to resume execution of #RMSVTSK. This routine dequeues (using a Receive Message instruction) the message and routes control (using the message key) to #RMSELIQ.

#RMSELIQ determines (according to the MPL rules) if any of the processes on the ineligible queue should be allowed to run. The TDEs of the selected processes are then dequeued from the ineligible queue and enqueued to the prime TDQ. A bring perform paging request is issued to get the process access group when the selected processes are executing on the prime TDQ.

#RMMIMWT, which invokes the service task, returns to #RMTDEAH. #RMTDEAH returns control (SVX), causing the receive instruction in #CFRWTO to be reissued. Since the access exception is now inhibited, the process is placed in a wait state (assuming the wait is still unsatisfied). From the time the purge page request is issued until the process actually enters the wait state, no pages in the access group (including IWA) are modified or purged.

Time Slice End

At time slice end (end of specified time interval), a process can either enter ineligible (for the current MPL) wait or remain in the current MPL set and continue execution. In the first case, processes in the machine are looped through the ineligible-wait TDQ; in the second case, they are looped on the prime TDQ.

When the processor time interval in the dispatcher interval timer equals 0, an internal microprogramming (IMP) exception is signaled. This exception is passed using an implicit SVL and control is transferred to the first-level exception handler (#SV00EXC).

#SV00EXC invokes the time slice end exception handler (#RMTSACQ) to replenish the time slice value (#RMRPLTS), updates the processor time used, and interrupts the current process causing the rest of time slice end processing to occur at the next System/38 instruction boundary (the process is scheduling an interrupt to itself).

Note: Time slice end can only occur when the current process is in execution and does not necessarily occur at System/38 instruction boundaries. The functions to be performed at time slice end must be done at System/38 instruction boundaries.

#RMTSACQ returns to the #SV00EXC which returns (SVX), and the current process continues execution until either a System/38 instruction boundary is reached or the process enters instruction wait. At such a boundary, the Supervisor Link Monitored instruction gives control to the interrupt function routing routines #SV1FBDY.

#SV1FBDY saves the routing bits in the exception CRE, clears the routing bits in PCBRES, saves the last waited-on SRQ, and inhibits process interruptions. Then, based on the time slice end routines bit, it invokes #RMTSEND that performs the following time slice end functions:

- Signals a machine interface event if the maximum allowed processor time is exceeded.
- Signals event if the time slice end without entering instruction wait processing option is specified.
- Checks if the current process should stay in the current MPL set. The current process is placed in the ineligible state if one of the following conditions is met:
 - The class or machine-wide MPL maximum values are exceeded. (This could arise from a Modify Process Attributes instruction, which changed the MPL class of the process, or from modify resource management controls.)
 - A process of higher or equal priority on the ineligible queue could run if the current process were made ineligible.

- If the process is to be placed in the ineligible state, #RMINEWT is invoked to perform the following functions:
 - Increment transition count.
 - Purge process access group if specified and required.
 - Send a message to the service task to initiate the selection of a process to run if an access group is to be purged.
 - Move current process to ineligible-wait TDQ (by disabling task dispatching, dequeuing it from the prime TDQ, enqueueing it to the ineligible queue and enabling task dispatching).
- If the process is to remain on the prime TDQ, #RMROBIN is invoked if the number of processes in the current MPL is greater than one.
 - #RMROBIN disables task dispatching and checks the priority of the next TDE.
 - If the priority of the next TDE is equal to that of the current TDE, then the current TDE is dequeued from and enqueued back to the prime TDQ.
 - Enables the task dispatching.

The process, if entered ineligible wait state, remains in ineligible wait state until it is selected by the service task, executing #RMSELIQ, to run in the current MPL. When the process is selected, #RMSELIQ dequeues it from the ineligible queue, enqueues it to the prime TDQ. The selected process is dispatched when it becomes the current process on the prime TDQ. The newly dispatched process resumes execution of module #RMINEWT, brings (using a perform paging request) the process access group if it had been purged, and returns to the caller (#RMTSEND or #RMIOCMP).

Leaving Instruction Wait

Wait Satisfied: A message is sent to the waited-upon SRQ, causing the RECM in #CFRWTO to be satisfied. #CFRWTO invokes #RMIOCMP, which determines if the process can run as part of the current MPL. If not, #RMIOCMP purges the process access group if it is in main storage and modify access state at instruction wait is specified, and then #RMINEWT is invoked to place the process in the ineligible state. If the process can run as part of the current MPL (or after #RMINEWT returns, indicating that the process has left the ineligible state and the process access group has been brought into main storage), #RMIOCMP brings the process access group if it exists and the modify access state at instruction wait is specified. Then, it returns to #CFRWTO, which cancels the time-out (#CFWTOC) and returns to the caller.

Interrupt a Process in an Instruction Wait: #RMINIPI (executing under the interrupting process) dequeues the waiting process from the SRQ, enables the SVLM, enables TDE access exceptions, and enqueues the process to the prime TDQ. As with an instruction wait, a TDE access exception occurs when the waiting-process starts to run on the prime TDQ. This causes control to be passed to #RMTDEAH, which issues a Supervisor Link Monitored instruction that invokes #SV1FBDY. #SV1FBDY invokes #RMIOCMP to determine if the process can enter the current MPL. If so, the process access group is brought into main storage and control is returned to #SV1FBDY. #SV1FBDY then invokes the pending interrupt servicing routines. Thus, the process in instruction wait state is interrupted to service the scheduled interrupt function. When interrupt service routines return control to #SV1FBDY, the routine returns control (SVX) to #RMTDEAH. #RMTDEAH then invokes #RMMIMWT, which operates as when entering an instruction wait to again place the process in the instruction wait state. (Note that if a message is placed on the SRQ during the initiate process interrupt processing, the process receives the awaited message first and leaves the wait state as with leaving the instruction wait. Then, the interruption is serviced at the next System/38 instruction boundary.)

Resource Management Service Task

The resource management service task is used to perform services that another task or process cannot do for itself.

#RMSVTSK waits on a queue (#RMSVQ) by issuing a Receive Message instruction to that queue. When a message is received, the value in the key field routes control to one of the following functions:

Key Value (Hex)	Function
0003	Select process from ineligible wait state (#RMSELIQ)
0005	Signal event for asynchronous lock wait time-out (#RMLKTO)
0007	Make more call/return elements available (#RMCREAC)
0009	Return TDEs and signal process terminated event (#PMFINAL)
8000	Resident bad page recovery (#RTIRTRY)

When the invoked modules return control to #RMSVTSK, the service task reexecutes the Receive (first message) instruction to execute or wait for the next requested function.

Resource Management Attribute Control

The attribute control functions support the following System/38 instructions:

- Materialize Resource Management Data
- Modify Resource Management Control

The Materialize Resource Management Data instruction is used to materialize processor utilization data, MPL data, auxiliary storage data, and main storage data. The Modify Resource Management Control instruction is used to alter selected attributes of MPL data, auxiliary storage data, and main storage pool data. The following list shows the process attributes that affect resource management. They can be materialized by the Materialize Process Attributes instruction:

- Priority
- Time slice value
- Time slice end signal event attribute
- Modify access state at instruction wait
- Modify access state at time slice end
- Process storage pool identification
- Maximum temporary auxiliary storage allowed
- Current temporary storage used¹
- Default time-out interval
- Maximum processor time allowed
- Total processor time used¹
- Process MPL class identification
- Number of changes from active state to the ineligible wait state¹
- Number of changes from active state to the instruction wait state¹
- Number of changes from the instruction wait state to the ineligible wait state¹
- The time that the latest state transition occurred

¹These attributes cannot be altered by the Modify Process Attributes instructions as these attributes reflect the statistics related to a process.

Machine resource management attributes are modified immediately. Process MPL attributes are modified under the target process through the process interruption mechanism. The effect of modification usually does not take place immediately. Modification of attributes are described in the following paragraphs.

Maximum MPL Value: The new maximum MPL value is greater than old maximum MPL value.

- **Machine-wide:** If there is any process in the ineligible wait state, service task is dispatched to select process(es) to run in the current MPL set.
- **MPL Class:** If there is any process in the ineligible wait state in that MPL class and if machine-wide current MPL is less than machine-wide maximum MPL, then the service task is dispatched to select ineligible process(es).

The new maximum MPL value is less than old maximum MPL value.

- **Machine-wide:** The effect is not immediate. No process in the current MPL set is forced out of the current MPL set even if the next maximum MPL is less than the current MPL. However, no process is allowed to enter the current MPL set until the current MPL is less than the new maximum MPL. Counts are corrected as processes enter instruction wait or reach time slice end.
- **MPL Class:** The effect is not immediate. The same algorithm described above for machine-wide MPL applies to MPL class.

Ineligible Threshold Value: The effect is not immediate. Ineligible threshold exceeded event can be signaled when either of the following occurs:

- An instruction wait satisfied or process is initiated but the process is ineligible for the current MPL set.
- The ineligible process selection algorithm indicates that there are additional ineligible processes.

Number of MPL Classes: Number of MPL classes cannot be modified. The number is initialized to 16.

When more than one MPL attribute is specified in modifying MPL controls, the order of modification is machine-wide maximum MPL, and then machine-wide ineligible threshold value.

MPL class attributes are modified by the same order, one class at a time. MPL class and machine-wide MPL attributes cannot be modified together.

Process MPL Parameters: Certain process attributes affecting resource management are modified as part of the Modify Process Attributes instruction. When the target of modification is not the current process, the target process is interrupted via the interrupt process mechanism and modification takes place under the target process at a System/38 instruction boundary. The actions are as follows:

- **Priority:** The function is performed by nucleus subroutine #RMMPRTY (packaged in module #RMTSACQ). The process disables task dispatching, dequeues itself from the prime TDQ, modifies its priority, re-enqueues itself back to the prime TDQ and enables task dispatching. The purpose is to make the current process the last among processes of equal priority on the prime TDQ, so that a processor-sharing effect is achieved.
- **Time Slice Value:** If the new time slice value is less than the machine minimum time slice value, the minimum is used and no exception is signaled. If the new time slice value is greater than the maximum allowed processor time value of the current process, the maximum value is used and no exception is signaled. (The machine minimum time slice value, a constant, is currently 50 ms.) The new time slice value takes effect at the next instruction wait or when the next time slice end is reached.
- **Maximum Processor Time Allowed:** If the new maximum (modified) process time allowed value is less than the amount of processor time used, the machine interface event, maximum allowed processor time expired, is signaled and the maximum allowed value is changed.

Normally, when the maximum processor time allowed is expanded, the VMC signals the event, replenishes the time slice with a full time slice value, and allows the process to continue with its program execution.

- **Process Access State at Instruction Wait:** The access state modification (such as purge) takes effect when the process next enters instruction wait.
- **Process Access State at Time Slice End:** The new value takes effect at the next time slice end.
- **Process MPL Class Identification:** The process is taken out of the current MPL set of the old MPL class and put into the current MPL of the new MPL class. Momentarily, the current MPL value of the new class could be larger than its maximum MPL value until a process in the class enters instruction wait or ineligible wait state.
- **Default Wait Time-Out Interval:** The modification takes effect at the next instruction wait.
- **Process Storage Pool Identification:** There is no immediate effect. Gradually, the process begins paging to the new pool.

Process Attribute Initialization

#PMINPR2 (initiate process, Part II) invokes #RMINIPR. This routine initializes certain resource management related process attributes (time slice, maximum processor time allowed, MPL class and storage pool identification), increments the machine and class total number of processes, and invokes #RMIOCMP to attempt to enter the process in the current MPL set. If not entered, #RMINEWT is called to place the process in the ineligible state. When the process leaves the ineligible state or if #RMIOCMP placed the process in the active state, return is made to #PMINPR2 and initiation continues.

Process Termination

As part of final process cleanup, #RMTERAT is invoked under the service task to decrement MPL counts, update processor time used by destroyed process since the last IMPL, and calls #RMSELIQ to select processes in the ineligible-wait TDQ to run in the current MPL.

Resource Management Attributes Modification

#PMMODF2 (modify process attribute) invokes #RMMODAT to modify process related resource management attributes. They are: priority, time slice value, maximum allowed processor time, MPL class identification and storage pool identification, and default wait time-out interval value.

The modification data is passed in the message on the process interruption SRQ. #RMMODAT dequeues the messages one at a time, modifies the specified attribute, until all messages for resource management attribute modification are dequeued. Then control is returned to the caller.

#RMMODC invokes #RMMODMP to modify MPL control attributes. They are: maximum MPL values and ineligible threshold values, either machine-wide or by MPL class.

#RMMATD invokes #RMMATMP to materialize MPL data. They are: maximum MPL, current MPL, ineligible threshold and number of ineligible processes either machine-wide or by MPL class, and transition counts by MPL class. #RMMATD also invokes #RMMATPU to materialize processor time used since last IMPL.

Access Group Control

The access group control functions support the following System/38 instructions:

- Create Access Group
- Materialize Access Group
- Destroy Access Group

An access group is an object that collects objects into a group that can be operated on by storage management to reduce auxiliary storage access. The access group is created as an object with a segment identifier and is allocated a block of contiguous space on auxiliary storage. Other objects can be allocated within the block. These objects have their own identifiers and can be accessed individually. Special directory information for the access group as a whole enables storage management to transfer all objects within the access group to and from main storage as a single unit. The access group is initially created containing no objects.

At process initiation, an access group can be specified. This access group should contain those objects used exclusively by the initiated process. This access group is called the process access group.

Normally, the specified access group is purged (written to auxiliary storage and the main storage frames occupied by the access group made available to other users) when a process leaves the active state, and brought into main storage when a process reenters the active state. This purge and bring can be overridden by setting to 0 the modify access state at instruction wait and time slice end process attributes. Additionally, any System/38 instruction that can place a process into an instruction wait contains a purge option. The purge only occurs if both the process and instruction attributes allow the purge.

Access Group Creation

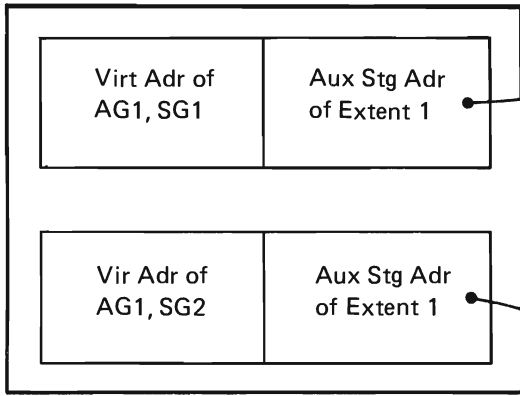
Access groups are created by module #RMCACG. This module validates the input creation template and then invokes storage management to create a temporary segment that will contain the encapsulated program architecture (EPA) and access group headers and the access group table of contents. #RMCACG again invokes storage management to create a temporary segment that will contain the segments of the access group.

#RMCACG then initializes the access group header and stores the address of the second segment in the access group header. The address of the first segment group is stored in the segment header of the second segment group.

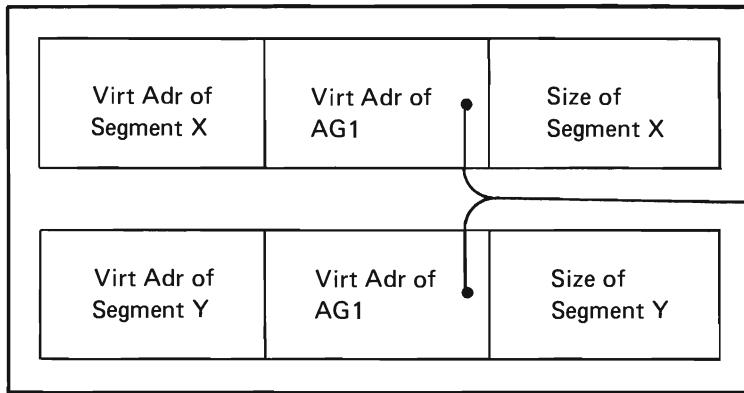
#RMCACG then stores the auxiliary storage address of the first page of the second segment group into the table of contents. If the access group is to contain an associated space, #RMCACG invokes storage management to create the space as a segment group in the access group.

The create access group module then builds the space pointer in the first segment group and the space pointer in the associated space. If the access group does not have an associated space, the space locator in the first segment group is set to 0's. #RMCACG then initializes fields in the EPA header and the building of the access is completed. Figure 38-4 shows an example of an access group and the storage management directories involved.

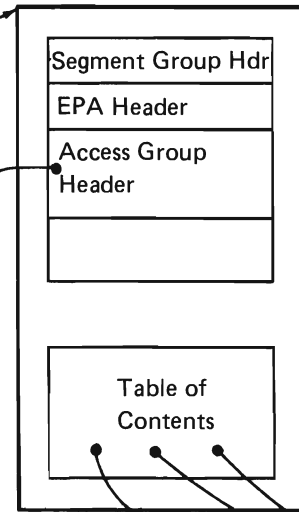
Temporary Directory



Access Group Member Directory



Access Group 1,
Segment 1,
Extent 1



Access Group 1,
Segment Group 2
Extent 1

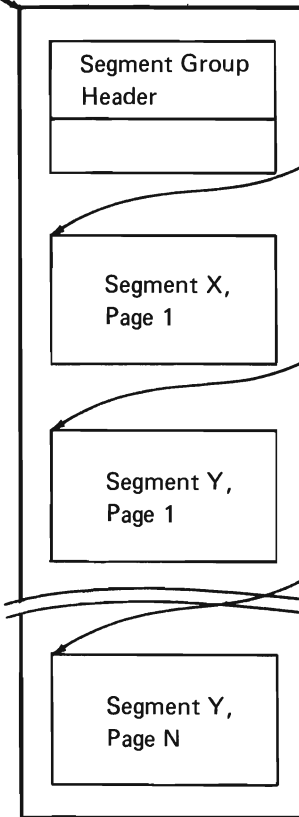


Figure 38-4. Access Group and Directories

Attribute Materialization

Module `#RMMACG` materializes the attributes of an access group. `#RMMACG` validates the size of the input materialization template and seizes the access group. `#RMMACG` materializes the access group information into an internal template, stores the access group size and available space in the access group into the template, and builds a system pointer to each object in the access group in the template provided in the instruction. `#RMMACG` then copies the fixed portion of the template and releases the access group.

Access Group Destruction

Module `#RMDACG` performs the destroy access group function. This module validates the system pointer to the access group and seizes the access group. `#RMDACG` invokes `#CFOCHKR` to perform other checks on the access group. `#RMDACG` then checks that the access group does not contain any objects.

`#RMDACG` then invokes `#CFDESTO` to begin the destroy process. If the access group has an associated space, `#RMDACG` invokes `#SMASM` to destroy the associated space segment group and null the pointer to the associated space. `#RMDACG` then invokes `#CFDESTO` to complete destruction of the access group.

Access Group Exception Processing

Module `#RMCACGC` is a component specific exception handler (CSEH) that is entered when an exception is signaled in `#RMCACG`. This exception handler destroys a partially created access group.

Module `#RMMACGC` is a CSEH that is entered when an exception is signaled in `#RMMACG`. This CSEH releases the access group.

DATA AREAS

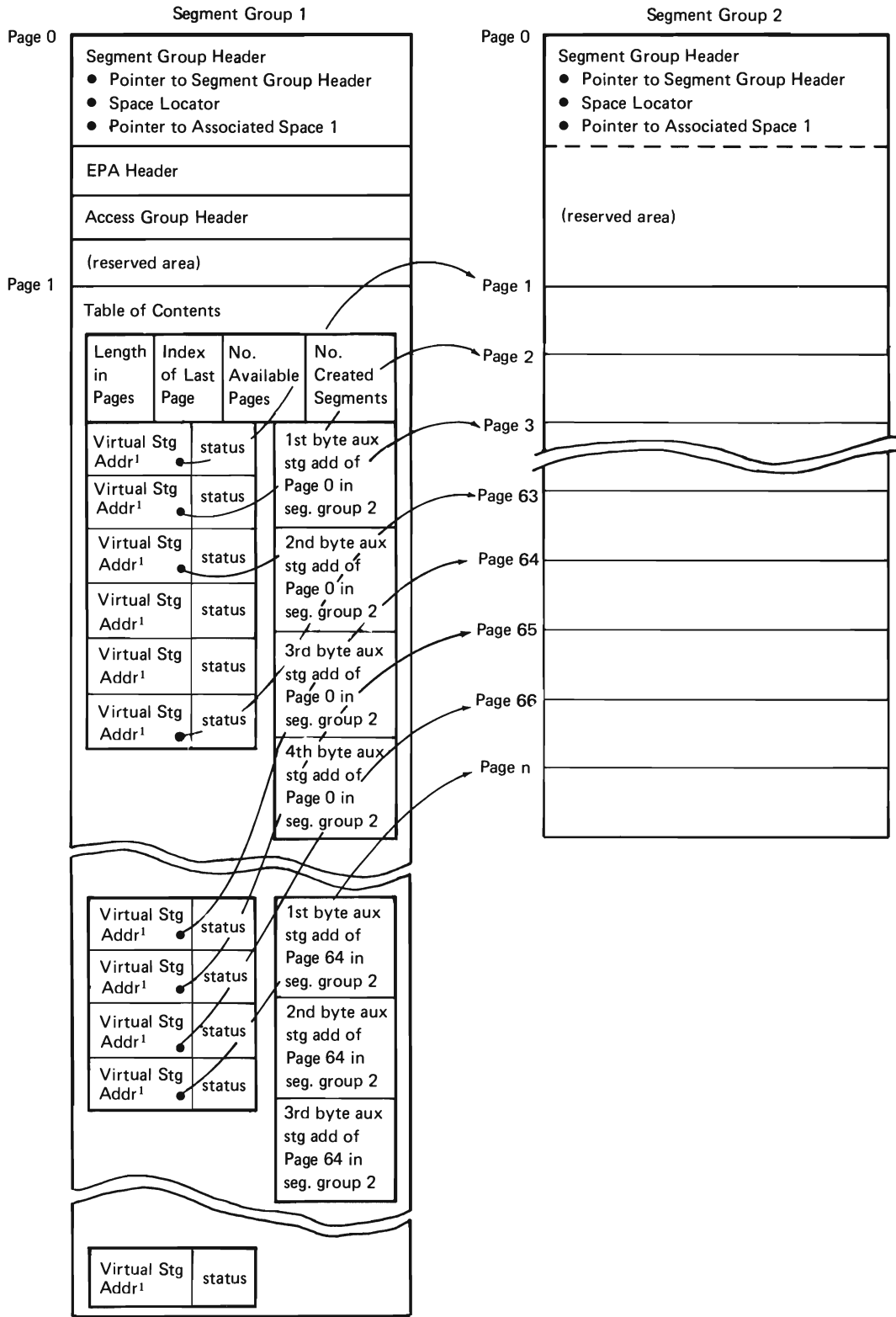
Access Group

An access group consists of two segment groups as shown in Figure 38-5. Segment group 1 contains a segment group header, the EPA header, the access group header, and a table of contents. The access group header only contains a pointer to the segment header of the second segment group. The table of contents starts on a page boundary following the access group header and is used to map the virtual address of a page of an object in the access group to its auxiliary storage address. Each page of the table of contents contains 64 entries, each 8 bytes long. The first entry of the first page consists of four 2-byte fields that contain the following information:

- Length of the access group segment in pages
- An index of the last allocated entry
- The number of available pages
- The number of segments in the access group

The remainder of the table of contents entries contain the 5 high-order bytes of the virtual address of a page of an object or a page in any virtual segment created into the access group. The last byte of the second, third, fourth, and fifth entries on each page of the table of contents contain the address of the auxiliary storage location where the virtual address in the first entry is mapped.

Segment group 2 contains the objects that are created into the access group. This segment group also contains the associated space of the access group if an associated space was created as a member of the access group. The first page of the second segment group contains the segment header (the remainder of this page is not used).



¹This field contains a null value if the associated space is not allocated to some object.

Figure 38-5. Access Group Structure

Clock Comparator Data Area (#RMCCDX)

This area contains the queue, counters, and data used by resource management timer services. The contents of this area are shown in Figure 38-6.

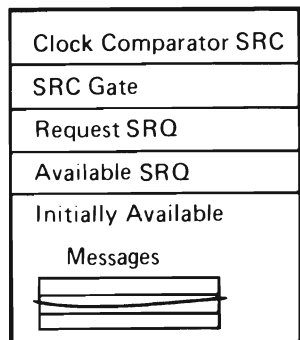


Figure 38-6. Clock Comparator Data Area (#RMCCDX)

Hold Hash Table

The hold hash table contains a series of halfword entries that provides an offset to an associated entry in the hold record area.

Hold Record Area

The hold record area contains a chain of entries. An entry is inserted in the hold record area when an object or data space entry is put into a lock or seize state. The format of each entry is as follows:

Hold Flags	Object Address	Object Identifier of Address Holding TDE	Chain Pointer	Data Base Area
------------	----------------	--	---------------	----------------

The data base area is not used by the grant and free instructions, but is used during some locking functions during data base operations.

Lock/Unlock Input Area

The lock/unlock input area is the input to the lock and unlock resource management functions. The contents of this area are shown in Figure 38-7. This area is used internally by the lock instruction processors and is the input for the lock horizontal interfaces.

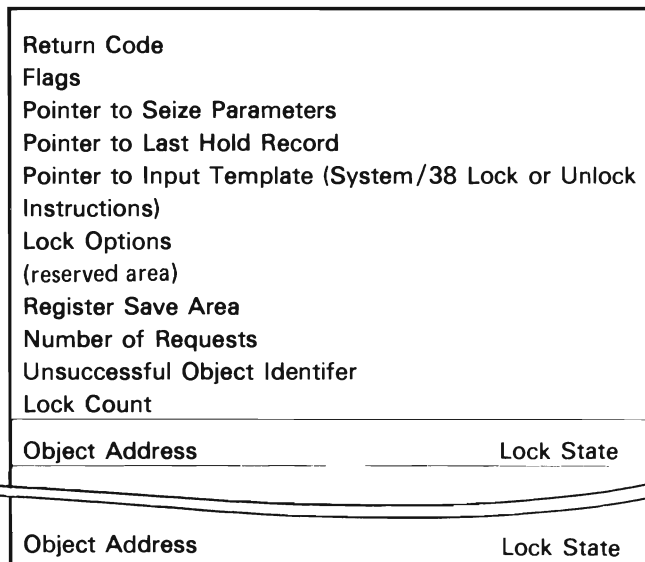


Figure 38-7. Lock/Unlock Area (ZZLKI)

Lock-Wait Data Area (#RMLKDX)

The lock-wait data area is used to maintain a list of waiters for a lock. The contents of #RMLKDX is shown in Figure 38-8. The waiting messages queue defines the objects that a process is awaiting. The waiting processes queue contains the processes that are synchronously waiting on the queue. When a process requests a lock-wait for an object, 3 bits of the object address are used to determine which of the eight areas is to be used. The 3 bits are the last 3 bits of the segment group number.

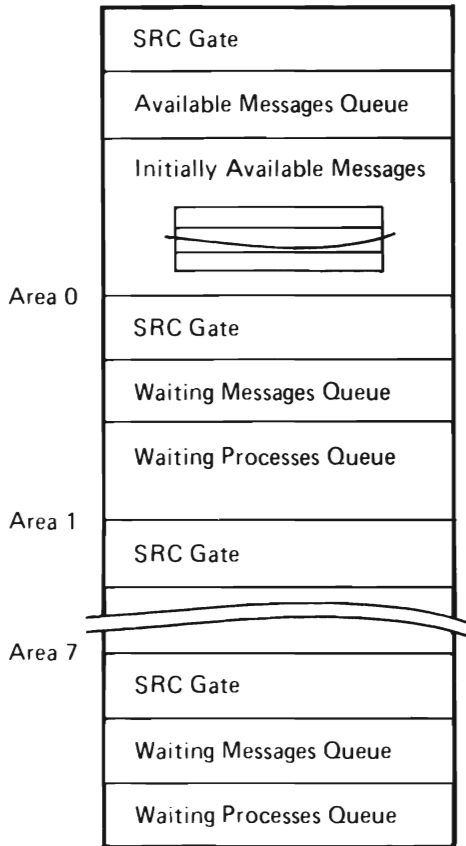


Figure 38-8. Lock-Wait Data Area (#RMLKDX)

Seize/Release Input Area

The seize/release input area maps the parameters for seize and release operations. The contents of this area are shown in Figure 38-9.

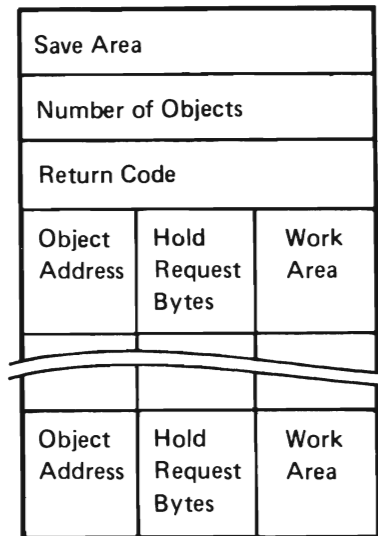


Figure 38-9. Seize/Release Input Area

Seize-Wait Data Area (#RMSZDX)

Area #RMSZDX is used during seize and release operations when waiting is required. The contents of this area are shown in Figure 38-10. The waiting messages queue defines the object that a TDE is waiting to seize. When a TDE must wait, 2 bits of the object address are used to determine which of the four areas is to be used.

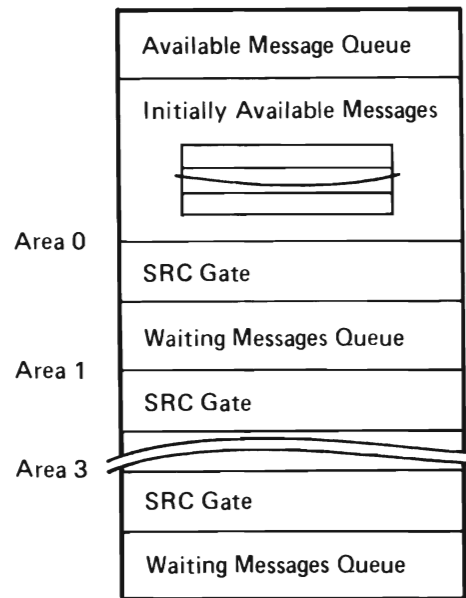


Figure 38-10. Seize-Wait Data Area (#RMSZDX)

STRUCTURE

The following is a list of the modules in resource management and the function that each module performs. The list also shows how the module is invoked.

#CFCTASK Create VMC or Microtask

Function: Establishes a VMC or a microtask and enqueues the TDE to the TDQ.

How Invoked: Other VMC components.

#CFDTASK Destroy Task

Function: Externally destroys a microtask or self-destroys a VMC task. This module will not destroy an external VMC task.

How Invoked: Other VMC components.

#CFRWTO Receive With Time-Out

Function: Logically performs a receive with time-out operation. Returns either user original message or a time-out indication.

How Invoked: Other VMC components.

#CFWTO Wait Time-out

Function: Builds a time-out request for the invoking program.

How Invoked: Other VMC components.

#CFWTOC Cancel Wait Time-Out Request

Function: This routine is invoked to cancel a time-out request. A specific request can be canceled or all outstanding requests for a process can be canceled.

How Invoked: Other VMC components.

#RMAEHS Access Exception Handler

Function: Receives control from the first-level exception handler (FLEH) for TDE, SRM, and SRQ access exception. This module runs under either VMC or user processes.

How Invoked: Other VMC components.

#RMBSYX Busy Exception Handler

Function: Causes a TDE getting a busy exception to enter a wait and then retries the failing operation.

How Invoked: Other VMC components.

#RMCACG Create Access Group

Function: Creates an access group.

How Invoked: Create Access Group instruction.

#RMCACGC Create Access Group CSEH

Function: Processes exceptions that occur during create access group operations.

How Invoked: Other VMC components.

#RMCCDX Clock Comparator Queues, Counters, and Data

Function: Provides storage for the queues, counters, and data used by the resource management clock comparator functions. This module does not contain executable code.

How Invoked: Not applicable.

#RMCCINT Clock Comparator Interrupt Handler

Function: When the clock comparator matches the time-of-day clock, this routine receives the count sent by the comparator. This routine then removes the first time-out request from this queue and performs the requested operation.

How Invoked: The Send Count instruction issued by HMC.

#RMCDUPC Create Duplicate Object CSEH

Function: Performs the required recovery for exceptions that occur during create duplicate object operations.

How Invoked: Other VMC components.

#RMCDUPO Create Duplicate Object

Function: Duplicates objects.

How Invoked: Other VMC components.

#RMCMBTC Restore Free TDE Queue and Available CRE Queue

Function: Combines any pairs of TDEs on the same page in the free TDE queue and deallocates the page. Combines any sets of four CREs on the same page in the free CRE queue and deallocates the page.

How Invoked: Other VMC components.

#RMCREAC CRE Access Exception Handler

Function: Adds four CREs to the end of the ACQ chain.

How Invoked: Within this component.

#RMDACG Destroy Access Group

Function: Destroys the designated access group.

How Invoked: Destroy Access Group instruction.

#RMDOULK Destroy Object Unlock

Function: After an object has been destroyed, this routine clears any locks on the object and dispatches any waiters to lock the object.

How Invoked: Other VMC components.

#RMDPULK Destroyed Process Unlock

Function: When a process is destroyed, this routine clears any outstanding lock waits and unlocks any locks held by the process.

How Invoked: Other VMC components.

#RMENOBJ Ensure Object

Function: Ensures that any changes made to a user object are recorded in the auxiliary storage.

How Invoked: Other VMC components.

#RMGFEX Grant-Free Exception Handler

Function: Processes the exceptions from the grant and free IMP instruction SCB, GHR, GHRF, FHRF, and FHR.

How Invoked: Other VMC components.

#RMGTREQ Get Clock Comparator Request Elements

Function: Creates additional clock comparator request elements.

How Invoked: Other VMC components.

#RMHHT Hold Hash Table

Function: Defines and initializes storage to be used by the hold/free instructions as a hash table. This module does not contain executable code.

How Invoked: Not applicable.

#RMHHXLK Transfer User Profile Lock when Process Created

Function: Transfers one lock from one process to another when the receiving process is being created.

How Invoked: Other VMC components.

#RMHLK Horizontal Interface to Lock

Function: Provides an interface for a VMC routine to obtain an implicit lock of an object or data space entry.

How Invoked: Other VMC components.

#RMHUNLK Horizontal Interface to Unlock

Function: Unlocks implicit locks. If any task or process is waiting to obtain a lock, dispatches the waiters.

How Invoked: Other VMC components.

#RMINEWT Ineligible Wait State

Function: Initiates the signaling of ineligible threshold exceeded events and the selection of the processes in the ineligible wait state if these bits are set. This module then moves the current process into the ineligible wait TDQ to wait for an open MPL slot.

How Invoked: Other VMC components.

#RMINIPI Initiate Process Interrupt

Function: Initiates process interrupt that interrupts another process of the current process itself.

How Invoked: Other VMC components.

#RMINIPR Resource Management Initiate Process Interface

Function: Starts time slicing and puts this process under MPL control.

How Invoked: Within this component.

#RMINIT Resource Management Initialization Routine

Function: Initializes resource management functions at IPL.

How Invoked: Other VMC components.

#RMIOCMP Transition of a Process from Wait

Function: Changes the state of a process from a wait state to the executing state in current MPL, or from wait state to ineligible wait state and then to current MPL set.

How Invoked: Other VMC components.

#RMLK Lock Instruction Processor

Function: Obtains object locks.

How Invoked: Lock Object instruction.

#RMLKAR Lock Asynchronous Wait Retry

Function: Asynchronously retries a lock instruction that failed previously because an object was locked in a conflicting state. If successful in obtaining the lock, signals an event.

How Invoked: Other VMC components.

#RMLKDX Lock Wait Data Areas

Function: Defines the data areas used when a lock wait is needed. This module does not contain executable code.

How Invoked: Not applicable.

#RMLKS Lock Space Location Processor

Function: Obtains space location locks.

How Invoked: Lock Space Location instruction.

#RMLKTO Asynchronous Lock Wait Time-Out

Function: Signals an event when an asynchronous lock wait exceeds the wait time specified.

How Invoked: Within this component.

#RMMACG Materialize Access Group Attributes

Function: Materializes the description of an access group and the identification of objects currently assigned to the access group.

How Invoked: Materialize Access Group Attributes instruction.

#RMMACGC Materialize Access Group CSEH

Function: Performs the required recovery for exceptions that occur during a materialize access group operation.

How Invoked: Other VMC components.

#RMMATD Materialize Resource Management Data

Function: Materializes user-specified data and machine maintained statistics concerning the control and use of machine resources.

How Invoked: Materialize Resource Management Data instruction.

#RMMATMP Materialize MPL Data (Subroutine)

Function: Obtains MPL data and stores it in RMMTMD area.

How Invoked: Within this component.

#RMMATPU Materialize Processor Utilization Data (Subroutine)

Function: Calculates processor time used by active and destroyed VMC tasks and user processes.

How Invoked: Within this component.

#RMMDTOD Modify Time-Of-Day

Function: Sets the time-of-day clock to the value presented in the instruction. Requests waiting for an interval of time must be modified so that the elapsed wait interval remains unchanged.

How Invoked: Modify Machine Attributes instruction.

#RMMIETO Backout Time-Out Messages at Terminate Invocation due to Unexpected Exceptions

Function: Restores time-out SRM when invocation of #CFRWTO is terminated unexpectedly.

How Invoked: Other VMC components.

#RMMIMWT Unconditional Wait MPL Control Routine

Function: Prepares the current process to enter a wait. This module adjusts the MPL and process attributes, invokes traces, initiates process selection when first entering the wait, and purges the process access group if conditions are met.

How Invoked: Within this component.

#RMMIWTO Handle Wait Time-Out

Function: Handles wait time-out prior to message received in the receive-with-time-out common function.

How Invoked: Other VMC components.

#RMMODAT Modify Attributes for Modify Process Attribute

Function: Modifies the specified resource management attributes.

How Invoked: Other VMC components.

#RMMODC Modify Resource Management Controls

Function: Modifies the specified resource management controls.

How Invoked: Other VMC components.

#RMMODMP Modify MPL Controls

Function: Modifies the specified values for MPL controls.

How Invoked: Within this component.

#RMMPLM Machine-Wide MPL and MPL Class Data Module

Function: Management of machine-wide MPL that governs the number of concurrently executing processes. MPL class data structure MPLC is also part of this module. This module does not contain executable code.

How Invoked: Not applicable.

#RMMTOLK Materialize Object Locks

Function: Materializes all locks on an object that are held or waited for.

How Invoked: Materialize Object Locks instruction.

#RMMTPLK Materialize Process Locks

Function: Materializes all locks that a process holds or is waiting for.

How Invoked: Materialize Process Locks instruction.

RMMTSLK Materialize Selected Locks

Function: Materializes all locks a process holds for selected objects or space locations.

How Invoked: Materialize Selected Locks instruction.

#RMMTTOD Materialize Time-Of-Day

Function: Stores the current time-of-day.

How Invoked: Materialize Machine Attributes instruction.

#RMPICRE Backout CREs

Function: Restores unserviced interrupt functions when exceptions cause prior invocation to be terminated.

How Invoked: Other VMC components.

#RMRCVLK Receive Transferred Locks

Function: Receives the locks that are transferred to this process.

How Invoked: Within this component.

#RMRLSA Release All Objects

Function: Releases all objects that are seized by the process invoking this routine.

How Invoked: Other VMC components.

#RMSACS Set Access State

Function: The instruction communicates a process usage or access speed requirements for a designated set of objects.

How Invoked: Set Access State instruction.

#RMSELIQ Select Process

Function: Selects a process in ineligible state to execute in the current MPL set.

How Invoked: Within this component.

#RMSLWT Synchronous Lock Wait Terminate

Function: Terminates a synchronous lock wait when the invocation in which the request was made no longer exists.

How Invoked: Other VMC components.

#RMSVTSK Resource Management Service Task

Function: Supports terminate process final phase code. Supports MPL control that selects and dispatches processes in the ineligible wait. Supports signal ineligible threshold exceeded event for MPL control functions. Supports asynchronous lock wait time-out.

How Invoked: Other VMC components at IPL and periodically by timer service.

#RMSZDX Seize Wait Data Area

Function: Maps the data areas used by seize and releases when waiting is required. This module does not contain executable code.

How Invoked: Other VMC components.

#RMTDEAH TDE Access Exception Handler

Function: Replenishes time slice value, supports process interrupt and interrupts at wait, and supports MPL control.

How Invoked: Within this component.

#RMTERAT Modify Attributes at Terminate Process

Function: Updates MPL attributes and processor time used, and dispatches process in ineligible wait state.

How Invoked: Other VMC components.

#RMTSACQ Time Slice End and ACQ Related Access Exception Handler

Function: Replenishes time slice, and updates processor time used. Indicates time slice end before completion status and indicates processor maximum time limit reached status. Analyzes exception code and invoke monitored CRE access exception handler.

How Invoked: Other VMC components.

#RMTSEND Time Slice End to Ineligible Wait or Continue

Function: Makes conditional transition of a process from the executing state to the ineligible wait state or causes it to remain in executing state at time slice end.

How Invoked: Within this component

#RMUGGR Unsuccessful Get MWS Gate Release

Function: When a get MWS encounters an exception, a gate must be released in some cases before presenting the exception. This allows other processing to continue. This routine releases the gates.

How Invoked: Within this component.

#RMUNLK Unlock Instruction

Function: Unlocks the specified object locks or cancels all asynchronous lock waits outstanding for the requesting process. If a specified lock is unlocked and a process is waiting to lock the object, this module dispatches the waiters.

How Invoked: Unlock instruction.

#RMUNLKS Unlock Space Location Processor

Function: Unlocks specified space locations.

How Invoked: Unlock Space Location instruction.

MACHINE-WIDE STORAGE

Machine-wide storage is a virtual storage area used by VMC modules for various free-storage applications. Storage is obtained by invoking #CFGTMWS at entry points #CFGTMWS and #CFGMWSR. Storage is freed by invoking entry points #CFFRMWS, #CFFMWSR, and #CFFMWSA. When an area is freed, the identification (normally the address of the current process control block) of the module returning the area must also be provided.

Virtual storage areas are supplied to the calling program in blocks of 32, 64, 128, 256, and 512 bytes. Requests for other sizes are rounded up to these sizes. Larger areas (513 through 65 024 bytes) are allocated and freed within the machine-wide storage (MWS) code by creating or destroying a segment identifier (SID) via calls to #SMSGCRT and #SMSGDES. Control information for the various blocks that have either been allocated or are available for allocation is kept in a control segment separate from the data areas given to the requesting program.

At IPL time, the resource management initialization module (#RMINIT) invokes #CFMWSIN to set up the storage blocks needed. Five expandable temporary SID groups are obtained for the five block sizes from 32 through 512 bytes, and 6 control segments (also expandable) are obtained for the control information for the five fixed sizes (corresponding to the five SID groups) and the large sizes (greater than 512 bytes).

#CFGTMWS allocates storage from the available areas. If space is not sufficient, the SID group (and the control segment if necessary) is extended to provide enough space to satisfy the call. For large sizes, the auxiliary storage management segment creation function is invoked to obtain a temporary segment which is given to the user, and the address and the identification of the owner are allocated.

In a free operation of up to 512 bytes, #CFFRMWS or #CFFMWSR or #CFGMWS (after checking the validity of the area to be returned), deallocates the area, making the space available for future get operations. For a free operation of areas greater than 512 bytes, #CFFRMWS or #CFFMWSR again makes a validity check, and then destroys the segment. The data in the control segment for the freed space is kept current.

DESTROY OBJECT (#CFDESTO)

When a System/38 destroy instruction (for a particular object) is issued, one of several instruction processors is invoked to destroy the specified object. After preliminary checking and processing is complete as appropriate for the object, it is necessary to perform operations that are common to all destroy operations.

#CFDESTO is invoked to perform these operations. #CFDESTO normally removes the address of the object from any contexts that reference the object, removes the authority to the object from any user profile(s), removes object ownership from the owning user profile, and then destroys the object segment identifier and an associated space (if any).

#CFDESTO then returns to the invoking instruction processor that performs any other functions required for the destroy.

#CFDESTO has two parts. The first part checks the addressing context (if any) and the owning user profile (for a permanent object) or the access group (for temporary objects) to ensure that neither is locked exclusively. The second part performs the actual deleting functions.

#CFDESTO also performs a cleanup function in the event that the invoking routine determines that the object cannot be destroyed.

Note: #CFDESTO is not called to destroy user profiles. User profiles are destroyed by #AUDESUP.

GET SPACE FROM IWA (#CFGIWA)

#CFGIWA allocates space from the invocation work area (IWA). The space is given contiguously beginning from the end of the area that is currently in use. The size of a request can be any value between 1 K and 64 K. Each request is rounded up to the next multiple of 16.

Other options control the allocation of an IWA area. These options can request that the allocated area be aligned on a page boundary or be contained within a page. An IWA allocation does not cross a segment boundary. The amount of IWA that can be allocated to a process is limited. An attempt to exceed this list results in automatic destruction of the process.

FREE SPACE FROM IWA (#CFFIWA)

#CFFIWA is an entry point in #CFEAOSE. #CFFIWA frees space in the IWA. Only storage belonging to the invocation can be freed. The first 24 bytes of an invocation area are not freed because these bytes contain forward and backward pointers. The number of bytes to be freed is rounded down to the next multiple of 16.

OBJECT CHECKER (#CFOCHKR)

#CFOCHKR validates a system pointer and optionally provides some diagnostics and serialization on the system object addressed through the system pointer. The basic checks performed by this function are as follows:

- Validate pointer type (must be a system pointer)
- Verify that the pointer is resolved (late binding)
- Verify the existence of the pointer
- Ensure that the reference object exists

This function optionally provides the following checks and functions:

- Object type check
- Authority check
- Seize the object
- Object damaged check
- Object suspended check
- Lock enforcement check

Any system pointer used by a VMC routine, whether it is passed in a System/38 instruction operand or as a pointer within a template, is diagnosed by the function. One or more pointers can be provided as input to #CFOCHKR in a single invocation.

One of the primary functions provided by #CFOCHKR is to test for a conflicting lock state and at the same time to serialize the activity on the system object.

REPORT OBJECT ON OBJECT RECOVERY LIST (#CFLOGRL)

#CFLOGRL performs the placement of entries on the object recovery list. Each entry identifies an object. Entries are placed on the list when they are found to be damaged (soft or hard), when they are not synchronized with a journal, when data space indexes are found to be invalidated during IPL, or when the entries are suspended during IPL (initial program load) recovery and during some run-time functions. #CFLOGRL sets a recovery flag to indicate the object recovery list is incomplete when discovered.

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.

access group table of contents: A list that describes the virtual and disk addresses of objects in an access group.

ACQ: See *available CRE queue*.

address list element: The address list element is an 8-byte object containing a virtual or virtual=real address to be used during page chaining operations. The address list element is a single element in a page chain address stack used during the processing of a function operation block command.

Advanced Program-to-Program Communications: 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.

AIPL: See *alternative initial program load*.

alternative initial program load: A process, when combined with the initial microprogram load sequence, that prepares the system for operation and installs CPF from the diskette magazine drive.

American National Standard Code for Information Interchange: (ANSI) The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communications systems, and associated equipment. The American National Standard Code for Information Interchange set consists of control characters and graphic characters.

American National Standards Institute: An organization sponsored by the Computer and Business Equipment Manufacturers Association for the purpose of establishing voluntary industry standards.

ANSI: See *American National Standards Institute*.

APPC: See *advanced program-to-program communication*.

ASCII: See *American National Standard Code for Information Interchange*.

ASDE: See *auxiliary storage directory entry*.

ASM: See *auxiliary storage management*.

auxiliary storage: All addressable storage space other than main storage. Auxiliary storage is located on the system's removable disk enclosures.

auxiliary storage directory entry: Contained in the permanent or temporary directory, the auxiliary storage directory entries are 11-, 16-, 21-, and 26-byte entries that map the disk addresses assigned to all permanent and temporary segments.

auxiliary storage management: Directs the allocation of auxiliary storage and maintains directories that map virtual addresses to disk locations. It also directs the allocation of space within access groups.

available CRE queue: The mechanism by which call/return elements are made available to the processor and eventually to a task dispatching element.

base address: A numeric value used as a reference in the calculation of addresses in the execution of a computer program.

base address register: (ISO) A register that holds a base address. Synonymous with base register.

basic information unit: In systems network architecture, the unit of data and control information that is passed between connection point managers. It consists of a request/response header followed by a request/response unit.

binary synchronous communications: A flexible form of line control that provides a protocol for communication between two stations.

break offset mapping table: An optional part of an encapsulated program template that consists of bit entries that provide mapping of the System/38 instructions to the high-level language source statement.

BSC: See *binary synchronous communications*.

BSTAT: Basic status.

buffer control list: The buffer control list is used to maintain the location and status of the input buffers. There is an entry in the buffer control list for each buffer. The entry contains a pointer to the buffer and the status of that buffer.

call/return element: A resident storage area used to save the status of a procedure during a supervisor linkage function.

CD: See *controller description*

commit: To cause all changes that were made to the data base file since the last commitment operation to become permanent and the records to be unlocked so they are available to other users.

commit block: A permanent object that serves as the structure to control commit/decommit within a process.

commitment control : A means of grouping file operations that allows the processing of a group of data base changes as a single unit or the removal of a group of data base changes as a single unit.

component-specific exception handler: An exception management function that is defined by a vertical microcode component to attempt to resolve an exception, replace the exception with a more meaningful one, or perform a cleanup function.

connection point manager: The systems network architecture component that provides a common mechanism by which session control, network control, and network addressable units communicate with their corresponding elements through the communications network. The units of information that the connection point manager receives from the network addressable units, session information to construct the transmission headers and request units.

Control Program Facility: The system support licensed program for the IBM System/38. It provides many functions that are fully integrated in the system such as work management, data base data management, job control, message handling, security, programming aids, and service.

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.

CPF: See *Control Program Facility*

CRC: See *cyclic redundancy check*.

CRE: See *call/return element*.

CSEH: See *component-specific exception handler*.

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.

cyclic redundancy check: In data communications, a method of error checking that is performed at both the receiving station and the transmitting station after a block check sequence has occurred.

data flow control: A data manipulation function that monitors and controls the exchange of requests and responses to and from the function manager according to the half-duplex send/receive mode of operation.

data link: (1) In data communications, the communications lines, modems, system consoles, work stations, and other communications equipment for the transmission of data between a receiving station and a transmitting station in a data network. (2) In data communications, the physical connection and the connection protocols between the host system and the communications control unit nodes by means of the host data channel.

data link control: The noninformational exchanges that set up, control, check, and terminate the information exchange(s) between two stations on a data link.

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 mapping table: An object that contains an identification entry for each data space that the cursor is over. The data spaces are in the same sequence as in the data space list supplied during program creation.

DFC: See *data flow control*.

DISC: An SDLC command for disconnect. The DISC command indicates that data cannot be received or transmitted.

disconnect mode: An SDLC response to disconnect mode. The disconnect mode response indicates that the receiving station is offline.

DKEY: Data space key table.

DKYT: Key field description table.

DLC: See *data link control*.

DMAP: See *data space mapping table*.

DSTAT: Device status.

EBCDIC: See *extended binary-coded decimal interchange code*.

ELLC: See *enhanced logical link control*.

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.

enhanced logical link control: An SNA logical link control protocol that allows the transfer of data link control information between two adjacent SNA nodes that are connected through an X.25-based packet-switched data network. This protocol provides enhanced error detection and recovery. Contrast with *physical services header* and *qualified logical link control*. Abbreviated ELLC.

EPA: Encapsulated program architecture.

EXCB: A machine index function.

exchange identification: The station identification sequence for establishing identity of the secondary station. The contents of this field establishes a unique identification of the physical unit that a controller description object is to represent. The block number and the specified identification are assigned by the manufacturer or installer for every physical unit. Each unit can identify itself with the exchange identification information.

extended binary-coded decimal interchange code: A set of 256 characters, each represented by 8 bits.

FAT: See *function address table*.

feedback record: A message placed on a queue indicating the status of a completed operation initiated by the Request I/O instruction.

first-level exception handler: An exception management function that receives control from the horizontal microcode when the horizontal microcode detects an error. The first-level exception handler then routes the exception to the appropriate common exception handler.

FOB: See *function operation block*.

format identification field: In systems network architecture, a field in a transmission header that defines the subsequent format of the header and the type of transmission header fields involved with transmission.

FSM: Finite state machine.

FSTAT: Functional status.

function address table: The function address table contains entries that describe functions being requested by the system control adapter to load and execute the system control adapter routines.

function operation block: One of five forms of the operation block that identify the operational unit and convey the command to be executed by the operational unit.

GB: See gigabyte.

gigabyte: One billion bytes.

HMC: See *horizontal microcode*.

horizontal microcode: The microcode that exhibits a high degree of parallelism of execution, controls the detailed state of the hardware, and supports the internal microprogramming instruction set.

I/O: Input/output.

I/O manager: A programming object that controls the flow of control information to and from an I/O unit.

IAR: See *instruction address register*.

ILC: See *instruction length count*.

IMP: See *internal microprogramming*.

IMPL: See *initial microprogram load*.

index control block: Defines the indexing operation to be performed, the argument to be used, the space for the result, and the location of the index.

initial microprogram load: The initialization procedure that causes the loading of the system microcode from disk or diskette to control storage.

initial program load: The initialization procedure that causes an operating system to start operations.

input/output controller: A functional unit in a data processing system that controls one or more units of peripheral equipment.

instruction address register: A register from whose contents the address of the next instruction is derived.

instruction length count: A 3-bit code that provides the length of the last instruction executed.

internal microprogramming: A set of instructions that are used as an internal communications link between the vertical microcode and the horizontal microcode.

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.

invocation work area: The area of storage where the execution of a program occurs.

IOC: See *input/output controller*.

IOM: See *I/O manager*.

IPL: See *initial program load*.

ISTAT: Immediate (or intermediate) status.

IWA: See *invocation work area*.

IXCB: See *index control block*.

journal: Refers to the journal port and the journal spaces attached to the journal port.

journal entry: The changes that are made to an object and are recorded in the journal spaces.

journal port: A system object used to link the journaled objects to the journal spaces.

journal space: Receives and records changes to objects specified as journaled objects.

journaled object: Object that has the changes made to it recorded in the journal spaces.

LEAR: See *lock*.

LENR: See *lock*.

light-emitting diode: A device that emits light for detection purposes.

link control block: The link control block is located in the invocation work area of the queue message router and is present when the task is active. It contains pointer, the program operation block, the operation request element, status indicators, and retry counters used for error recovery.

LKB: See *link control block*.

LLC: See *logical link control*.

load/dump: A function that enables the user to save (back up) certain permanent objects by dumping these objects to a load/dump media (such as diskettes) and, then, load (restore) these objects when needed.

local session identification: In systems network architecture, a field in a local address format (FID3) that identifies both the origin and destination network addressable units of a given session.

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 link control: See also *exhanced logical link control, qualified logical link control, and physical services header*.

logical unit: In systems network architecture, one of three types of network addressable units. It is the port through which an end user accesses function management in order to communicate with another end user. It is also the port through which the end user accesses the services provided by the system service control point. It must be capable of supporting at least two sessions—one with the system services control point, and one with another logical unit. It may be capable of supporting many sessions with other logical units.

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.

LSRD: See *lock*.

LSRO: See *lock*.

LSUP: See *lock*.

LU: See *logical unit*.

LUD: See *logical unit description*.

LUSTAT: Logical Unit Status (SNA command).

machine check: A type of exception that indicates a malfunction on the machine.

machine check logout buffer: An area in main storage that is reserved for the reporting of machine checks by the hardware machine check handler. This area contains data concerning the cause of the machine check and the environment existing when the machine check occurred.

machine communications area: The assigned storage locations, which contain control information required for VMC objects to communicate with each other.

machine initialization status record: Used by VMC components to store information relating to the status of the machine following an initial program load or an initial microprogram load.

machine services control point: The machine component that provides services and coordinates the processing of supervisory services.

machine-wide storage: A virtual storage area used by VMC modules for various free-storage applications.

main storage: All storage in a computer from which instructions can be executed directly.

main storage management: Manages the functions necessary to read data from or write data to main storage.

materialization definition template: A part of an encapsulated program template that contains a copy of the input program template if the program was specified as observable in the Create Program instruction.

MCA: See *machine communications area*.

MCLB: Machine Check logout buffer.

MCR: Machine configuration record.

MDT: Materialization definition template.

MFCU: Multi-function card unit.

MISR: See *machine initialization status record*.

MPL: See *multiprogramming level*.

MSCP: See *machine services control point*.

MSM: See *main storage management*.

multiprogramming level: The number of processes currently executing.

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 extend 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 *name resolution*.

ND: See *network description*.

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.

NRL: See *name resolution list*.

object definition table: A part of a program definition template that defines the program objects associated with the instructions in its instruction stream.

object mapping table: A part of an encapsulated program template that consists of a variable-length vector of 6-byte entries. The number of entries corresponds to the number of object definition table directory vector entries. Each object mapping table entry provides a location mapping for the object defined by an associated object definition table directory vector entry.

ODT: See *object definition table*.

ODT directory vector: One of the components of the object definition table. The ODT directory vector consists of a series of 4-byte entries. These entries are referred to by the operands of machine interface instructions and provide a description of the program object. The object definition table entry string is used to complete the description when it cannot be completely described with the 4-byte ODT directory vector.

ODV: ODT directory vector.

OES: ODT entry string.

OMT: See *object mapping table*.

OP: See *operation program*.

operation program: A set of operation blocks placed in storage and executed together prior to any response.

operation request element: An internal microprogramming message, placed on an operational unit queue, to cause an I/O operation. It consists of a standard internal microprogramming queue element header, a status field, and an operation block.

operational unit: An I/O device or source of asynchronous events together with an operational unit task that controls the device or the event.

operational unit queue: The queue upon which operation request elements are placed by the source/sink component below the machine interface. There is one operational unit queue for each operational unit.

operational unit task: A microcode task that exists for each operational unit that performs operations such as operation block execution and command completion functions. The operational unit task services the operational unit queue and I/O events.

ORE: See *operation request element*.

ORP: Optional response poll.

OU: See *operational unit*.

PASA: See *process automatic storage area*.

path control: In systems network architecture, one of the components of the transmission subsystem, and one of two components of the common network. It is responsible for managing the sharing of data link resources of the common network and for routing basic information units through it. It is aware of the location of network addressable units in the network and of the paths between them. It maps the basic transmission units, handled by transmission control, into path information units, and then into basic transmission units that are passed between path control and data link control. The unit of control information built by the sending path control component and interpreted by the receiving path control component is a transmission header.

path information unit: In systems network architecture, the unit of transmission consisting of a transmission header and either a basic information unit or a basic information unit segment.

PCB: See *process control block*.

PCS: See *process control space*.

PDE: See *primary directory entry*.

PDT: Process definition template.

PE: See *phase-encoding*.

PEM: See *program event monitor*.

phase modulation recording: A magnetic recording in which each storage cell is divided into two regions which are magnetized in opposite senses; the sequence of these senses indicates whether the binary character represented is zero. Synonymous with phase encoding.

phase-encoding: Synonym for phase modulation recording.

physical services header: One of three logical link control protocols used by IBM SNA DTEs. Physical services header provides adjacent node services. Contrast with *enhanced logical link control* and *qualified logical link control*. Abbreviated PSH.

physical unit: In systems network architecture, one of three types of network addressable units; a physical unit is associated with each node that has been defined to a system services control point. A physical unit controls the resources local to its associated node. The system services control point establishes a session with the physical unit as part of the bring-up process.

PIWA: Process invocation work area.

POB: Pointer operation block.

PPR: Perform paging request.

PRE: Paging request element.

primary directory element: An entry in the primary directory.

primary directory entry: A list of entries in which each entry contains the virtual address and the status of a page frame in main storage.

process automatic storage area: A space that is used for automatic program allocation when a program is invoked.

process control block: A storage area used to map storage areas and provide control areas. It also contains a pointer to the name resolution list.

process control space: A system object used to support the execution of a process and as a means of addressing a process.

process static storage area: A space that is used for static program object allocation during program activation.

program event monitor: The processor comparing the initial byte of the instructions to determine if they fall within the range of the program event monitor start and stop addresses.

program operation block: One of five forms of the operation block that is used in an operation request element when an operation program is to be executed.

PSH: See *physical services header*.

PSSA: See *process static storage area*.

PU: See *physical unit*.

QCT: See *queue control table*.

queue: A system object consisting of an ordered list of messages that communicates information to other processes.

queue control table: A table, accessed by microcode and machine product code, that controls I/O operations. There is one table for each operational unit.

RAM: Random access memory.

RD: See *request descriptor*.

read-only storage: Storage that can be read but not modified.

request descriptor: A field of the source/sink request. It contains the command operations for the I/O device or the message description to be sent to a communication device. The request descriptor field is uniquely defined for each type of I/O device on the system.

request/response header: In systems network architecture, a control field, attached to a request/response unit, that specifies the type of request/response unit being transmitted—request or response—and contains control information associated with that request/response unit. It is used by sending and receiving connection point managers to coordinate data traffic between network addressable units.

request/response unit: In systems network architecture, the basic unit of information entering and exiting the transmission subsystem. It may contain data, acknowledgement of data, commands that control the flow of data through the network, or responses to commands.

ROS: See *read-only storage*.

routing table: Supports the systems network architecture transmission subsystem. This table contains fields that support path control, connection point manager, data flow control, and logical units.

SCA: See *system control adapter*.

SDLC: See *synchronous data link control*.

second-level exception handler: An exception management function that invokes the third-level exception handler to process vertical microcode exceptions, invokes the exception generator to process translated code exceptions, or invokes a default exception handler to process the exception.

segment identifier: Bits 0 through 23 of a virtual address.

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.

send/receive counter: The internal microprogramming instruction object used to exchange intertask information and to synchronize the flow of control between tasks; a count field used for control but no messages are enqueued.

send/receive message: An internal microprogramming instruction object that contains a message and may be enqueued to a send/receive queue.

send/receive queue: An internal microprogramming instruction object that is used to exchange intertask information to synchronize the flow of control between tasks.

service order table: In the network control program, the list of telecommunication devices on a multipoint line (or point-to-point line where the terminal has multiple components) in the order in which they are to be serviced by the network control program.

SFD: Service function driver.

SID: See *segment identifier*.

SLEH: See *second-level exception handler*.

SM: Service monitor.

SMVT: Storage management vector table.

SNA: See *systems network architecture*.

SOT: See *service order table*.

source/sink active device list: A control block that contains the information maintained and used by various machine services control point routines.

source/sink data area: A space that contains the data areas (I/O buffers) associated with the operations requested by the request descriptors within the source/sink request.

source/sink request: The operand of a source/sink Request I/O instruction (MI) 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.

SRC: See *send/receive counter*.

SRM: See *send/receive message*.

SRQ: See *send/receive queue*.

SSCP: See *system services control point*.

SSD: See *source/sink data area*.

SSR: See *source/sink request*.

standard character stream: A method of transferring print data, carriage control, and print position information from the System/38 to a printer.

supervisor linkage: The method by which internal microprogramming procedure switching is accomplished within a task and the method by which internal microprogramming exceptions are reported.

supervisory services: The network control program supervisor code that provides miscellaneous services, such as the communication adapter interface, starting channel output, controlling timer operations, and data manipulation and utility services.

SVL: See *supervisor linkage*.

SVLM: Supervisor link monitor.

SVX: Supervisor exit.

synchronous data link control: A discipline for the management of information transfer over a data communications channel. Transmission exchanges can be duplex or half duplex; the communications channel configuration can be point-to-point, multipoint, or loop. SDLC includes comprehensive detection and recovery procedures, at the data link level, for transmission errors that can be introduced by the data communications channel.

system control adapter: An interface used in conjunction with the CE/operator panel for initiating and monitoring the system during system initiation.

system services control point: In systems network architecture, a network addressable unit that provides configuration, maintenance, and session services via a set of command processors (network services) supporting physical units and logical units. The system services control point must be in session with each logical unit and each physical unit for which it provides these services. It also provides services for the network operators or administrators who control the configuration. The system services control point is commonly located at a host node.

systems network architecture: The total description of the logical structure, formats, protocols, and operations sequences for transmitting information units through the communications system.

task: (1) A semi-independent unit of work that can be performed concurrently with other tasks and requires coordination with other tasks only at certain points within the execution. (2) Units of work activated by the task dispatcher.

task dispatching element: An internal microprogramming object used to identify a task and the attributes associated with that task.

task dispatching queue: An internal microprogramming object used by the task dispatcher to allocate processor time to the dispatchable tasks in the system.

TDE: See *task dispatching element*.

TDQ: See *task dispatching queue*.

third-level exception handler: An exception management function that invokes vertical microcode component-specific exception handler routines and performs an action based on the status of the exception after the component-specific exception handler routines have executed.

TLEH: See *third-level exception handler*.

TOD: Time of day.

transmission header: In systems network architecture, a control field attached to a basic information unit or to a basic information unit segment, and used by path control. It is created by the sending path control component and interpreted by the receiving path control component.

transmission subsystem: In systems network architecture, the innermost layer of the communication systems. It provides the control in each session to route and move data units between network addressable units and their interconnecting paths. Its three constituent parts are data link control, path control, and transmission control.

TS: See *transmission subsystem*.

UPS: Uninterruptible power supply.

vertical microcode: Microcode that defines logical operations on data, is primarily sequential in execution and supports the System/38 machine instruction set.

virtual address: The address of a storage location within virtual storage.

virtual storage: The combination of main storage and auxiliary storage, treated as a single addressable unit.

VMC: See *vertical microcode*.

volume table of contents: An area on a disk or diskette that describes the location, size, and other characteristics of each data file on the disk or diskette.

#CFCTASK 37-9
 #CFDAMEH 36-17
 #CFDESTO A-1
 #CFFIWA A-2
 #CFGIWA A-2
 #CFGTMWS A-1
 #CFLOGRL A-2
 #CFMIDEH 36-11
 #CFOCHKR A-2
 #CFRWTO 38-12, 38-15
 #CFSGEVT 35-3
 #CFSPECF 36-13
 #CFVFYEH 36-12
 #CFWTO 38-11
 #CRWTOC 38-12
 #DBXBLKY 2-16
 #DBXFDSh 2-5
 #DBXFIXH 2-5
 #DBXMUSE 2-4
 #DBXRINX 2-5
 #DOMATIA 12-2
 #DOMTINS 12-2
 #DOMTPTL 12-1
 #DOMTPTR 12-1
 #DOMTSOB 12-1
 #DOTRCEX 12-2
 #DOTRCLC 12-2
 #DOTRIEV 12-2
 #DOTRINS 12-1
 #DOTRINX 12-2
 #DOTRNV 12-2
 #DOTRSEV 12-1
 #EMDBLED 35-2
 #EMEBLED 35-2
 #EMMNEVT 35-1
 #EMREVTD 35-2
 #EMSGEVT 35-2
 #EMTSEVT 35-1
 #IXCRTIX 3-2
 #IXDESTIX 3-2
 #IXINSEN 3-4
 #IXMAIN 3-3
 #IXMATAT 3-4
 #IXMODII 3-4
 #JOAPPLY 4-5
 #JOCRTJP 4-5
 #JOCRTJS 4-5
 #JODESJP 4-6
 #JODESJS 4-6
 #JOINIT 4-8
 #JOJOB 4-6
 #JOMATJO 4-7
 #JOMATJP 4-6
 #JOMATJS 4-6
 #JOMATOA 4-7
 #JOMODJP 4-7
 #JORETEN 4-7
 #JOURDAT 4-6
 #JOURNL 4-6
 #LDPREP 26-3
 #LOCHPCI 21-6
 #LOCHRCI 21-6
 #LOSHIOM 21-3
 #MSAELRB 27-12
 #MSCABAN 27-2
 #MSCDIAL 27-2
 #MSCDRDC 27-13
 #MSCPTSK 27-1
 #MSCVOFF 27-2
 #MSERRLG 22-1
 #PMDTYPR 37-9
 #PMEXGEN 37-9
 #PMINIT 10-2
 #PMINPR1 37-4
 #PMINPR2 37-4
 #PMMATER 37-5
 #PMMODF1 37-5
 #PMMODF2 37-5
 #RMCACG 38-26, 38-28
 #RMCCDX 38-30
 #RMGFEX 36-14
 #RMGTREQ 38-12
 #RMINIPI 38-15
 #RMLKDX 38-31
 #RMMACG 38-28
 #RMMDTOD 38-11
 #RMMTTOD 38-11
 #RMSDTSK 38-22
 #RMSZDX 38-31
 #RMWTOE 38-12
 #RTVMCIR 10-1
 #SMACDIR 8-6
 #SMAGCRT 7-4
 #SMAGDES 7-4
 #SMAGEXT 7-4
 #SMMSIT 8-14
 #SMSGCRT 7-4
 #SMSGDES 7-5
 #SMSGEXT 7-5
 #SMSGTRC 7-5
 #SMSHTDN 7-7
 #SOCRT 6-1
 #SODES 6-3
 #SOMAT 6-1
 #SOMOD 6-1
 #SVE8PPR 8-1

#SV1FBDY 38-15
 #SV2DCRT 7-2
 #TPBRQIC 20-4
 #TPLACIN 24-5
 #TPLACOT 24-5
 #TPLALK1 24-4
 #TPLAUDL 24-5
 #TPLCOUT 24-5
 #TPLDAKO 24-13
 #TPLDCIN 24-5
 #TPLESCR 24-5
 #TPLSRMR 24-6
 #TPLWDNS 24-5
 #TPLXIDO 24-5
 #TPLXID1 24-5
 #TPLXID2 24-5
 #TPNRQIO 29-7
 #TPRALKO 24-4
 #TPRQIO 20-4
 #TPSFMX 24-10
 #TPSRQID 31-5
 #TXXIOM 34-7
 #XCODEGN 18-4
 #XDATAGN 18-4
 #XEH 18-4
 #XINIT 18-4
 #XLATOR 18-1
 #XODTSC 18-4
 #XREGOPT 18-4
 #XSCAN 18-4
 #XTERM 18-4

A

access exception handler 36-10
 access exceptions 38-1
 access group
 control 38-25
 create segment in 7-4
 creation 38-26
 data area 38-28
 data areas 7-9, 8-14
 destroy segment in 7-4
 destruction 38-28
 exception processing 38-28
 extent segment in 7-4
 member directory 8-14
 data area 7-11, 8-14
 operations, serialization of 7-4
 processing 7-3
 table of contents 8-14
 table of contents data area 7-12
 truncate segment in 7-4
 address translation (page fault) exception handler 36-5
 addressing, virtual storage 0-4
 advanced program-to-program communications 19-1

advanced program-to-program communications (continued)
 error logging 19-9
 I/O support 19-7
 instruction support 19-4
 SNA support 19-6
 alternate IMPL 0-14
 apply journal changes 4-5
 ASM locks 7-6
 asynchronous message handling 27-8
 attribute materialization 38-28
 authorization enforcement
 object authorization 14-2
 privileged instruction 14-2
 resource usage 14-2
 special authorization 14-2
 authorization management
 data areas 14-3
 modules 14-5
 object function 0-7, 14-1
 auxiliary storage
 initialization 7-7
 management 7-1
 management locks 7-6
 management modules 7-13
 space accounting 7-3
 auxiliary storage usage field 14-4

B

binary synchronous communications
 I/O manager 20-1
 data areas 20-6
 modules 20-7
 bring/purge access group 8-11
 busy exception 38-2
 busy exception handling 36-9

C

call/return element availability 38-2
 cancel event monitor 35-2
 cancel trace 12-2
 chain conflict end-of-chain and second chain exception handle 36-14
 channel error message 21-7
 channel event processing function 21-3
 channel I/O management 21-1
 channel I/O management modules 21-7
 channel vary on/off message 21-7
 cleanup routine, internal 25-5
 clock comparator data area 38-30
 commit management 0-2, 1-1
 commit 1-3
 commitment control 1-1

- commit management (continued)
 - cursor support 1-5
 - data areas 1-7
 - data space index 1-2
 - data space support 1-6
 - decommit 1-3
 - journal support 1-4
 - record locking 1-2
 - recovery, IMPL 1-2
 - structure 1-9
- common function A-1, 0-13
- components, relationship of 0-14
- connect 24-6
- context data areas 15-2
- context management
 - data areas 15-2
 - introduction 15-1
 - modules 15-6
 - object function 0-7
- controller description data area 23-3, 29-9
- CRE availability 38-2
- create
 - independent index 3-2
 - instruction processors 23-1
 - process control space 37-4
 - segment 7-4
 - segment in access group 7-4
 - space 6-1
 - task 37-9
- create journal port 4-5
- create journal space 4-5
- CSEH request block 36-17
- cursor 2-23
- cursor recovery 2-7
- cursors 2-2

D

- damage CSEH 36-17
- data area
 - access group 38-28
 - member directory 7-11, 8-14
 - table of contents 7-12
 - clock comparator 38-30
 - context 15-2
 - controller description 23-3, 29-9
 - CSEH request block 36-17
 - encapsulated program 18-5
 - encapsulated program architecture header 15-6
 - error log request 22-1
 - event index 35-4
 - function address table 32-5
 - function operation block 29-11
 - hold hash table 38-30
 - hold record 38-30
 - invocation control block 14-4

- data area (continued)
 - link control block 28-7
 - link control block (LKB) 24-15
 - lock/unlock input 38-30
 - lock-wait 38-31
 - logical unit description 23-3, 29-9
 - lookaside directory 8-14
 - lookaside table 29-11
 - machine communication area 15-6
 - machine initialization status record (MISR) 10-6
 - machine-wide storage 24-15
 - monitor event template 35-4
 - name resolution list 15-6
 - native control block 29-10
 - network description 23-4
 - operation request
 - element/program
 - operation block 29-11
 - OU/ND table 23-4
 - paging request element 8-15
 - permanent directory 7-10
 - primary directory 8-15
 - process automatic storage area 17-3
 - process control block
 - (nonresident) 35-4
 - (PCB) 14-4
 - (resident) 35-4
 - process control space 37-10
 - process definition template 37-10
 - process static storage area 17-5
 - program template 18-5
 - routing table 29-10, 31-7
 - sector headers 7-13, 8-15
 - seize/release input 38-31
 - seize-wait 38-31
 - service order table 28-7
 - signal event messages 35-4
 - source/sink active device list 27-24
 - static directory 8-14
 - station control block 31-6
 - storage management vector 8-15
 - storage management vector table 7-13
 - storage pools 8-16
 - storage queues (search and change) 8-16
 - system pointer 14-4
 - task dispatching element 15-6, 35-4
 - temporary directory 7-10
 - trace table 12-2
 - user message for SCA IOM 32-4
 - user profile 14-3
 - VMC communications area (VCA) 10-6
- data areas 19-1
 - access group 7-9, 8-14
 - authorization management 14-3
 - binary synchronous communications
 - I/O manager 20-6
 - context management 15-2
 - controller description table 20-6

- data areas (continued)
 - data base management 2-7
 - error log 22-2
 - event management 35-4
 - free space directory 7-9
 - half-session control block 30-10
 - independent index 3-4
 - index control block 9-6
 - index description template 3-5
 - initialization/termination management 10-6
 - journal management 4-8
 - link control block 20-6
 - machine check management 11-3
 - machine observation management 12-2
 - machine services control point 27-24
 - multi-leaving telecommunications access method I/O manager 28-7
 - network architecture control block 30-9
 - operation request element 20-6
 - permanent directory 8-14
 - process management 37-10
 - program execution management 17-3
 - program management 18-5
 - queue management 5-2
 - resource management 38-28
 - service order table 20-6
 - SNA output areas 30-10
 - source/sink 0-10, 23-3, 29-11
 - space object management 6-3
 - station I/O management 31-6
 - synchronous data link control input areas 30-10
 - synchronous data link control primary I/O manager 24-15
 - synchronous data link control secondary I/O manager 30-9
 - system control adapter I/O management 32-4
 - temporary directory 8-14
- data base management
 - data function 0-2, 2-1
 - data sharing 2-2
 - modules 2-28
 - recovery and IPL 2-6
 - structure 2-28
- data base objects, locking 2-2
- data function 0-2
 - data base management 0-2, 2-1
 - independent index management 0-3
 - queue management 0-3
 - space object management 0-3
- data pointer resolution 15-1
- data sharing 2-2
- data space 2-8
 - index 2-12
 - index recovery 2-7

- data space (continued)
 - management data areas 2-7
 - recovery 2-7
- destroy
 - independent index 3-2
 - instruction processors 23-1
 - object A-1
 - process control space 37-4
 - segment 7-5
 - segment in access group 7-4
 - space 6-3
 - task 37-9
- destroy journal part 4-6
- destroy journal space 4-6
- directory lookup 8-6
- directory recovery, storage management 7-7
- disable event monitor 35-2
- dump journal space 4-7
- dump space management 6-2
- dumping objects 26-4

E

- effective address overflow exception handler 36-5
- emulation, 3270 33-1
- enable event monitor 35-2
- encapsulated program architecture (EPA) header data area 15-6
- encapsulated program data area 18-5
- error checking the I/O request 26-4
- error log
 - data areas 22-1
 - modules 22-2
 - request data area 22-1
- establishing a CSEH 36-15
- event index data area 35-4
- event management
 - data areas 35-4
 - introduction 35-1
 - modules 35-5
 - supervisor and control function 0-11
- exception backout 38-16
- exception handler modules 36-18
- exception management 0-11, 36-1
- exchange bring/clear 8-11
- extend segment 7-5
- extend wait-request queue 38-12
- extent segment in access group 7-4

F

- FAT 32-5
- FBR 0-10
- feedback record 0-10
- find entry 9-4
- find/remove independent index entry 3-3
- first-level exception handler 36-4
- FLEH 36-4
- free space directory data areas 7-9
- free space from IWA A-2
- function
 - common 0-13
 - data 0-3, 2-1
 - internal machine 0-4, 7-1
 - machine support 0-6, 38-1
 - object 0-7
 - objects 14-1
 - program control 0-8, 17-1
 - source/sink 0-9
 - supervisor and control 0-11, 35-1
 - vertical microcode 0-1
- function address table data area 32-5
- function operation block data area 29-11

G

- get space from IWA A-2

H

- hold hash table data area 38-30
- hold record data area 38-30

I

- I/O managers 0-9
- I/O pending 8-10
- ICB 14-4
- IMPL 0-14, 10-1
 - alternate 0-14
- in-use table 2-27
- independent index
 - attributes, materialize 3-4
 - create 3-2
 - data areas 3-4
 - destroy 3-2
 - entry find/remove 3-3
 - entry, insert 3-4
 - management 0-3, 3-1
 - modify 3-4
 - modules 3-5

- index control block 9-6
- index control block, data areas 9-6
- index description template data areas 3-5
- index structure 9-2
- initial microprogram load (IMPL) 10-1
- initial program load (IPL) 10-3
- initialization 38-10
 - auxiliary storage 7-7
 - main storage 8-13
- initialization/termination management 0-6, 10-1
 - data areas 10-6
- initiate process 37-4
- initiate process interrupt 38-15
- insert entry 9-5
- insert independent index entry 3-4
- instruction processors 0-9
- instructions, trace 12-1
- interface, synchronous data link control secondary I/O managers 30-1
- intermediate mapping 2-12
- internal cleanup routine 25-5
- internal machine functions 0-4, 7-1
- Internal machine functions, machine index management 0-6
- internal machine functions, storage management 0-4
- internal microprogram load 0-14
- internal trap 24-14
- interrupt function router 38-15
- invalid segment group address exception handler 36-15
- invocation control block (ICB) data area 14-4
- invocation destruction 17-3
- invoking ASM functions 7-2
- IOM queue 0-10
- IPL 10-3
- IPL recovery 4-7
- IPL synchronization 4-8

J

- journal data 4-6
- journal management 4-1
- journal management data areas 4-8
 - object functions 0-7
 - structure 4-9
- journal modules 4-9
- journal object 4-6

K

key specification area 2-15

L

leaving instruction wait 38-22
link control block 34-3
link control block (LKB) data area 24-15
link control block data area 28-7
load/dump
 load/dump activation 26-4
 load/dump and suspend 2-4
 load/dump management 26-1
 modules 26-8
load journal space 4-7
loading objects 26-6
local I/O manager modules 25-5
local I/O managers 25-1
lock enforcement 38-6
lock entries 2-3
lock support 38-6
lock/unlock input data area 38-30
lock-wait data area 38-31
locking data base objects 2-2
locks, auxiliary storage management 7-6
logical unit description data area 23-3,
 29-9
lookaside directory 8-14
 data area 8-14
lookaside table 29-11

M

machine check management
 data areas 11-3
 introduction 11-1
 machine support function 0-6
machine communication area (MCA) data
 area 15-6
machine configuration record (MCR) 0-10
machine index management 0-6, 9-1
machine index management modules 9-7
machine index structure 9-7
machine indexes, operations on 9-4
machine initialization management
 modules 10-6
machine initialization status record
 (MISR) 10-6
machine initialization status
 record (MISR) data areas 10-6

machine observation management
 data areas 12-2
 introduction 12-1
 machine support function 0-7
 modules 12-2
machine processing, terminate 10-6
machine services control point
 data areas 27-5
 introduction 27-1
 modules 27-18
 source/sink function 0-9
machine support function
 initialization/termination management 0-6
 introduction 0-6
 machine check management 0-6
 machine observation management 0-7
 resource management 38-1
 service and installation management 0-7
machine-wide storage A-1
 data area 24-15
main storage
 initialization 8-13
 management 8-1
 modules 8-17
 paging function 8-1
 paging function tasks 8-1
mainline processing 8-4
management
 authorization 0-7, 14-1
 auxiliary storage 7-1
 context 0-7, 15-1
 data base 0-3, 2-1
 event 0-11, 35-1
 exception 0-11, 36-1
 independent index 0-3, 3-1
 initialization/termination 0-7
 machine check 0-6, 11-1
 machine index 0-6, 9-1
 machine observation 0-7, 12-1
 main storage 8-1
 process 0-11, 37-1
 program 0-8, 18-1
 program execution 0-8, 17-1
 queue 0-3, 5-1
 resource 0-12, 38-1
 service and installation 0-7, 13-1
 space object 0-3, 6-1
 storage 0-4
 system control adapter I/O 32-1
managers, I/O 0-9
materialization, program 18-4
materialize
 independent index attributes 3-4
 instruction processors 23-1
 invocation 12-2
 machine attributes (time-of-day) 38-11
 point locations 12-1
 pointer 12-1
 process attributes 37-5
 space attributes 6-1

- materialize (continued)
 - system object 12-1
- materialize journal port attributes 4-6
- materialize journal space attributes 4-6
- materialize journaled object
 - attributes 4-7
- materialize journaled objects 4-7
- MCA 15-6
- MCR 0-10
- member directory, access group 7-11
- message elements, queue 5-3
- MISR 10-6
- modify
 - controller description
 - (asynchronous) 27-7
 - controller description
 - (synchronous) 27-2
 - instruction processors 23-1
 - logical unit description
 - (synchronous) 27-5
 - LUD processing 26-6
 - machine attributes (time-of-day) 38-11
 - network description (synchronous) 27-5
 - process attributes 37-5
 - space attributes 6-1
- modify independent index 3-4
- modify journal port 4-7
- modify process event mask 35-3
- modules
 - authorization management 14-5
 - auxiliary storage management 7-13
 - channel I/O management 21-7
 - context management 15-6
 - data base management 2-28
 - error log 22-2
 - event management 35-5
 - exception handler 36-18
 - independent index management 3-5
 - journal 4-9
 - load/dump management 26-8
 - local I/O manager 25-5
 - machine index management 9-7
 - machine initialization management 10-6
 - machine observation management 12-2
 - machine services control point 27-26
 - main storage management 8-17
 - native I/O management 29-11
 - process management 37-11
 - program execution management 17-7
 - program management 18-7
 - queue management 5-3
 - resource management 38-32
 - service and installation
 - management 13-1
 - source/sink instruction processor 23-4
 - space object management 6-3
 - station I/O management 31-8
 - synchronous data link control I/O
 - manager 24-16

- modules (continued)
 - synchronous data link control
 - secondary I/O managers 30-12
 - system control adapter I/O
 - management 32-5
- monitor event 35-2
- monitor event template data area 35-4
- MSCP 27-1
- MSCP functions 29-7
- MSM 8-1
- MSM locks 8-12
- multi-leaving telecommunications access
 - method I/O manager
 - data areas 28-7
 - introduction 28-1
 - modules 28-7
 - structures 28-7
- multiprogramming level support 38-17

N

- name resolution list (NRL) data area 15-6
- native control block data area 29-10
- native I/O management
 - I/O support 29-7
 - introduction 29-1
 - modules 29-11
 - SNA support 29-4
 - System/38 instruction support 29-3
- network description data area 23-4
- NRL 15-6
- numeric exception handler 36-14

O

- object authorization 14-2
- object checker A-2
- object function, journal management 4-1
- object functions 0-7, 14-1
- object functions, authorization
 - management 0-7
- object functions, context management 0-7
- object functions, journal management 0-7
- object list data area 4-9
- object serialization 38-3
- operation request element (ORE) 0-10
- operation request element/program
 - operation block 29-11
- operational unit (OU) queue 0-10
- operations on machine indexes
 - find entry 9-4
 - insert entry 9-5
 - remove entry 9-6

ORE 0-10
OU 0-10
OU/ND table data area 23-4
overview, vertical microcode 0-1

P

page out task 8-7
page replacement 8-6
pageable channel IOM 21-6
paging function
 main storage management 8-1
 MSM 8-1
 tasks 8-1
paging request element
 data area 8-15
 processing 8-3
PCB 14-4, 15-6
PCS 37-10
PDT 37-10
permanent directory 8-14
permanent directory data area 7-10
permanent directory data areas 8-14
point locations, materialize 12-1
pointer tags 8-12
pointer, materialize 12-1
PRE 8-3, 8-15
PRE processing 8-3
primary directory 8-15
primary directory data area 8-15
privileged instruction 14-2
process
 attribute initialization 38-24
 automatic storage area data area 17-3
 control block (nonresident) data area 35-4
 control block (PCB) data area 14-4
 control block (resident) data area 35-4
 control block data area 15-6
 control space data area 37-10
 definition template data area 37-10
 interruption 38-12
 management 0-11, 37-1
 data areas 37-10
 modules 37-11
 static storage area data area 17-5
 termination 38-25
processing
 access group 7-3
 mainline 8-4
 PRE 8-3
processors
 create instruction processors 23-1
 destroy instruction 23-1
 instruction 0-9
 materialize instruction 23-1
 modify instruction 23-1
 request I/O 23-1

processors (continued)
 source/sink instruction 23-1
program
 creation 18-1
 deactivation 17-3
 destruction 18-4
 execution management 0-8, 17-1
 data areas 17-3
 modules 17-7
 invocation 17-2
 management 0-8, 18-1
 data areas 18-5
 modules 18-7
 materialization 18-4
 observability 18-4
 template data area 18-5
 user exit 2-22
program activation 17-1
program control functions 0-11, 17-1
 program execution management 0-8
 program management 0-8
program-to-program communications 19-1

Q

QCT 0-10
queue control table (QCT) 0-10
queue management
 data areas 5-2
 data function 0-3
 introduction 5-1
 modules 5-3
 recovery 5-2
queue message elements 5-3

R

read data store 24-14
receive message with wait-time-out 38-15
receive-wait time-out handler 38-16
receive with time-out 38-12
recoverable error processing 26-8
recovered object list 10-7
recovery
 cursor 2-7
 data space 2-7
 data space index 2-7
 queue management 5-2
recovery initialization 0-7, 16-1
relationship of components
 alternate IMPL 0-14
 internal microprogram load (IMPL) 0-14
 vertical microcode and the System/38 instruction set 0-15
remove entry 9-6
report object on object recovery A-2

- request I/O processors 23-1
- resident channel IOM 21-6
- resource management
 - attribute control 38-22
 - attributes modification 38-25
 - data areas 38-28
 - introduction 38-1
 - modules 38-32
 - service task 38-22
 - supervisor and control function 0-12
- resource usage 14-2
- resume process 37-7
- retrieve event data 35-2
- retrieve journal entries 4-7
- routing table data area 29-10, 31-7

S

- SCA 32-1
- SDLC autopoll flow 24-12
- SDLC I/O manager 24-1
- second-level exception handler 36-10
- sector headers 8-15
- sector headers data area 7-13, 8-15
- segment
 - create 7-4
 - destroy 7-5
 - extend 7-5
 - truncate 7-5
- seize/release 38-5
- seize/release input data area 38-31
- seize-wait data area 38-31
- send/receive messages 0-10, 26-6
- serialization (non-access group) 7-6
- serialization of access group
 - operations 7-4
- serialization of process
 - interruption 38-16
- service and installation
 - management 0-7, 13-1
- service and installation management
 - modules 13-1
- service order table data area 28-7
- session control 29-6
- shutdown, storage management 7-7
- signal event data 35-2
- signal event messages data area 35-4
- SLEH 36-10
- source/sink active device list 0-10
- source/sink active device list data
 - area 27-24
- source/sink data areas 0-10, 23-3, 29-11
- source/sink functions
 - I/O managers 0-9
 - instruction processors 0-9
 - machine services control point 0-9
 - source/sink data areas 0-10

- source/sink instruction
 - instruction processors 23-1
 - processor modules 23-4
- source/sink OU/ND table 0-10
- space
 - create 6-1
 - destroy 6-3
- space accounting, auxiliary storage 7-3
- space attributes, materialize 6-1
- space attributes, modify 6-1
- space object management 0-2, 6-1
- space object management data areas 6-3
- space object management modules 6-3
- special authorization 14-2
- specification default exception
 - handler 36-13
- start/halt device function 21-1
- static directory 8-14
- static directory data area 8-14
- station control block data area 31-6
- storage addressing, virtual 0-4
- storage management
 - directory recovery 7-7
 - internal machine function 0-4
 - shutdown 7-7
 - vector 8-15
 - vector data area 8-15
 - vector table (SMVT) data area 7-13
- storage pools 8-16
- storage pools data area 8-16
- storage queues (search and change) 8-16
- storage queues (search and change) data
 - area 8-16
- structure 16-1, 19-11
- structure, data base management 2-28
- structure, journal management 4-9
- supervisor and control functions
 - event management 0-11
 - exception management 0-11
 - process management 0-11
 - resource management 0-12
- suspend journal space 4-7
- suspend process 37-5
- synchronous data link control primary
 - line I/O manager
 - contact 24-6
 - data areas 24-15
 - discontact 24-6
 - error flow 24-13
 - introduction 24-1
 - modules 24-16
 - normal flow 24-4
 - SDLC autopoll flow 24-12
 - test 24-10
 - vary off 24-13
- synchronous data link control secondary
 - I/O managers
 - error logging 30-9
 - I/O support 30-8
 - interface 30-1

synchronous data link control secondary
 I/O managers (continued)
 introduction 30-1
 SNA support 30-5
 System/38 instruction support 30-4
 synchronous data link control station
 I/O management
 data areas 31-6
 error logging 31-6
 I/O support 31-5
 introduction 31-1
 modules 31-8
 SNA support 31-4
 System/38 instruction support 31-3
 system control adapter I/O management
 data areas 32-4
 introduction 32-1
 modules 32-5
 system object, materialize 12-1
 system pointer data area 14-4
 system-wide journal list data area 4-8
 System/38 instruction wait 38-20

T

table of contents, access group 7-12
 task dispatching element (TDE) 0-14
 task dispatching element data area 15-6,
 35-4
 task dispatching queue (TDQ) 0-14
 TDE 0-14, 15-6, 35-4
 TDE access exception handler 38-15
 TDE availability 38-3
 TDE time-out 38-2
 TDQ 0-14
 temporary directory 8-14
 temporary directory data area 7-10
 temporary directory data areas 8-14
 terminate machine processing 10-6
 terminate process 37-7
 test event 35-2
 third level exception handler 36-14
 time-slice end 38-20
 time-slice end and CRE exception
 handler 36-5
 timer services 38-11
 TLEH 36-14
 trace and cancel trace invocations 12-2
 trace instructions 12-1
 trace table data area 12-2
 translate tables 2-12
 translation, 3270 33-5
 trap table 34-5
 truncate segment 7-5
 truncate segment in access group 7-4

U

user exit program 2-22
 user message for SCA IOM 32-4
 user profile data area 14-3
 user profile index and table 14-5
 user profile recovery 14-4

V

VCA 10-6
 verify exception handler 36-12
 vertical microcode and the
 System/38 instruction set 0-15
 vertical microcode functions 0-1
 vertical microcode overview 0-1
 virtual storage addressing 0-4
 VMC communications area data area 10-6
 VMC default exception handler 36-11

W

wait on event 35-2
 wait time-out 38-11
 wait time-out cancel 38-12
 wait time-out for event 38-12

X

X.25 IOM 34-1

3270 emulation 33-1
 BSC 33-1
 manager 33-1
 3270 emulation data areas 33-6
 3270 emulation structure 33-7
 3270 translation 33-5

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®

SY21-0889-5



SY21-0889-5