

Program Product

**MVS/Extended Architecture
System Programming
Library:
System Modifications**

MVS/System Product:

JES3 Version 2	5665-291
JES2 Version 2	5740-XC6

IBM

Third Edition (March, 1987)

This is a major revision of, and obsoletes, GC28-1152-1. See the Summary of Amendments following the Contents for a summary of the changes made to this manual.

This edition applies to Version 2 Release 2.0 of MVS/System Product (5665-291 and 5740-XC6) and to all subsequent releases until indicated in new editions or technical newsletters. Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 System/370 Bibliography*, GC20-0001, for the editions that are applicable and current.

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

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

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department D58, Building 921-2, PO Box 390, Poughkeepsie, N.Y. 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

SPL: System Modifications is intended for the people who set up and maintain the system software for a data processing center running under IBM's Multiple Virtual Storage/Extended Architecture (MVS/XA) operating system. Although there are many titles for these people, (system manager, systems analyst, installation designer, system programmer), we will call them *system programmers*. Thus, *SPL: System Modifications* is written for the system programmer who has extensive experience with MVS, and who is familiar with its basic concepts.

The purpose of *SPL: System Modifications* is to help you decide where and how to modify the operating system. This book presents ways to customize the system.

Modifying the Operating System: An Overview

Should the software shipped by IBM as the MVS/XA operating system, not accommodate every device, application, or software add-on included in your installation, you can customize the system to fit your needs. As your installation grows and changes to meet new demands and emphasis, you can change the customization and adapt it to your new needs.

As you customize the system, keep in mind the importance of protecting the integrity of the system code and your own code. The System Integrity section of Volume One of *MVS/XA SPL System Macros and Facilities* tells how to maintain the integrity of MVS/XA and your programs.

SPL: System Modifications can help you decide the best way to customize the MVS/XA operating system.

The Customization Task

The MVS/XA operating system includes many points at which you, the system programmer, can assume control of the system code so you can modify it. These *control points*, listed in order of increasing refinement, are:

- **MVS Configuration Program** which defines the installation's I/O configuration to the operating system through SYS1.PARMLIB members.
- **System initialization**, which activates the operating system.
- **Job or step execution**, which activates or starts an installation application is accomplished

Within each of these are means of control that become more varied and complex as the degree of refinement increases. The IBM MVS/XA library of publications documents the various tools and means available to the system programmer for customizing the system. Usually the information is specific to the control points or to the system components. *SPL: System Modifications* presents the tools and means you can use to customize the operating system in the context of three major system programming tasks:

- Modifying the operating system to fit the devices in your installation
- Modifying the operating system to fit the work your installation does
- Accommodating the operating system to other software, especially subsystems, you want to use

All of these tasks can be addressed at each of the control points listed earlier; the means available to accomplish them depend on the control point.

You can use the following tools to customize your system at system initialization (IPL):

- System parameters included in members of SYS1.PARMLIB
- Replaceable modules in LPALIB, LINKLIB, and other system libraries
- System and private macro libraries
- Operator commands
- User exit routines (inserted at IPL; executed at job/step execution)

You can use the following tools to customize your system at job/step execution:

- Installation defaults for JCL and JES2/JES3 parameters
- Operator commands
- User exit routines

In addition to these, the MVS system includes many *system services* that provide flexibility and allow you to control installation processing at all three control points.

How This Book is Organized

This book has four parts:

- **Part I, Modifying the System to Fit the Devices in Your Installation**
- **Part II, Modifying the System to Fit Your Applications,**
- **Part III, Fitting Your Subsystems into the System,**
- **Part IV, MVS System Services,**

The chapters within each part of the book present the control points where you can make modifications, the tools IBM supplies to help you modify the operating system, and the alternatives you might consider.

Even though the topics are presented as separate tasks, they overlap and affect each other at many points: your device configuration will be influenced by your applications and job mix; the subsystems you use or develop will depend on both your device configuration and your applications.

Following is a summary of the chapters in this book.

- **Part I, Modifying the System to Fit the Devices in Your Installation,** includes the following:
 - **Chapter 1, Writing Unit Information Modules for the MVS Configuration Program**
 - **Chapter 2, Allocation Considerations**

Chapter 2 includes discussions of

 - **The Device Preference Table**
 - **The Eligible Device Table**
 - **The Volume Attribute List**
 - **Controlling GRS Requests**
- **Part II, Modifying the System to Fit Your Applications,** includes the following:
 - **Chapter 3, Limiting User Region Size**
 - **Chapter 4, Assigning Special Program Properties to Applications**
 - **Chapter 5, Creating Your Own Resource Managers**
 - **Chapter 6, Executing DAT-off Code in MVS/XA**
 - **Chapter 7, Controlling System Messages and the System Log**
 - **Chapter 8, Modifying the Master JCL**
 - **Chapter 9, Customizing the System Trace Table**
- **Part III, Fitting Your Subsystems into the System,** presents the following:
 - **Chapter 10, Defining Secondary Subsystems to MVS**
 - **Chapter 11, The Subsystem Affinity Service**

- **Part IV, MVS System Services**, presents several services provided by MVS/XA to help you accommodate the system to your installation and its applications. These system services are:
 - **Chapter 12, The Unit Verification Service**
 - **Chapter 13, The Hot I/O Detection Table**
 - **Chapter 14, The Internal Reader Facility**
 - **Chapter 15, The External Writer**
 - **Chapter 16, The Virtual Fetch Service**
 - **Chapter 17, The Dumping Services (ABEND and SVC Dumps)**

Other MVS/XA Publications

Throughout this book, we give general information about methods and control points, but do not try to cover a topic exhaustively unless it is not covered elsewhere in the MVS/XA library. Rather, we refer you to other publications where you can find more detailed information about a subject.

Note: The actual references in this book, use shortened book titles: “MVS/Extended Architecture” becomes “MVS/XA,” and “System Programming Library” becomes “SPL.”

Following are the publications cited in this book:

- *MVS/Extended Architecture Checkpoint/Restart*
- *MVS/Extended Architecture Data Management Macro Instructions*
- *MVS/Extended Architecture Data Management Services*
- *MVS/Extended Architecture Debugging Handbook Volumes 1-5*
- *MVS/Extended Architecture Diagnostic Techniques*
- *MVS/Extended Architecture JCL User's Guide*
- *MVS/Extended Architecture JCL Reference*
- *MVS/Extended Architecture JES3 Diagnosis*
- *MVS/Extended Architecture Message Library: Routing and Descriptor Codes*
- *MVS/Extended Architecture Message Library: System Messages*
- *MVS/Extended Architecture MVS Configuration Guide and Reference*
- *MVS/Extended Architecture Operations: JES3 Commands*
- *MVS/Extended Architecture Operations: System Commands*
- *MVS/Extended Architecture System Logic Library* Multiple volumes; Volume 1 contains order numbers for the other volumes and a general table of contents.
- *MVS/Extended Architecture System Programming Library: Data Management*
- *MVS/Extended Architecture System Programming Library: Initialization and Tuning*
- *MVS/Extended Architecture System Programming Library: JES2 Initialization and Tuning*
- *MVS/Extended Architecture System Programming Library: JES3 Initialization and Tuning*
- *MVS/Extended Architecture System Programming Library: Service Aids*
- *MVS/Extended Architecture System Programming Library: Supervisor Services and Macro Instructions*
- *MVS/Extended Architecture Installation: System Generation*
- *MVS/Extended Architecture System Programming Library: System Macros and Facilities*. Two volumes
- *MVS/Extended Architecture System Programming Library: System Management Facilities*
- *MVS/Extended Architecture System Programming Library: User Exits*
- *MVS Extended Architecture Interactive Problem Control System (IPCS) Planning and Customization*
- *MVS Extended Architecture Interactive Problem Control System (IPCS) User's Guide*
- *MVS Extended Architecture Interactive Problem Control System (IPCS) Command Reference*

Additional References for New Users of MVS/XA

If you are new to the MVS/XA operating system and library, you may want to read the following books, to learn how the system differs from its predecessors, and to get a general introduction to its concepts.

- *MVS/Extended Architecture Conversion Notebook* discusses the differences between MVS/XA and its immediate predecessor, MVS/370.
- *MVS/Extended Architecture System Programming Library: 31-Bit Addressing* presents the concept of extended addressing in terms of the IBM operating system. It includes information on planning for and writing programs in a 31-bit environment.

Contents

Modifying the System to Fit the Devices in Your Installation

Chapter 1. Writing Unit Information Modules for the MVS Configuration Program 1-1

How MVSCP Uses UIMs	1-2
Initialization Processing	1-2
IODEVICE Statement Processing	1-2
End-of-Data Processing	1-2
UIM Data Areas	1-3
UIM Service Routines	1-4
CBPADIT - Builds DITs Service Routine	1-5
CBPDDCT - Build DCT Service Routine	1-6
CBPIDFT - Build the DFTs Service Routine	1-7
CBPIFEAT - Device Features Checker Service Routine	1-8
CBPIGETM - Getmain Service Routine	1-9
CBPIPARM - IODEVICE Parameter Checker Service Routine	1-10
UIM Processing Logic	1-11
Considerations for UIM Processing	1-12
UIM Macros	1-14
CBPYDIP	1-16
CBPZCPVT	1-17
CBPZDCP	1-18
CBPZDFP	1-19
CBPZDIAG	1-20
CBPZFCP	1-22
CBPZGETM	1-23
CBPZIODV	1-24
CBPZITRH	1-25
CBPZLOG	1-26
CBPZLOGR	1-28
CBPZPCP	1-29
CBPZPPDS	1-30
CBPZUCA	1-31
Device Support Modules and Macros	1-33
IOSDDT - Device Descriptor Table Build Macro	1-34
IOSDMLT - Module Lists Table Macro	1-37
Writing a UIM	1-38
Naming a UIM	1-38
UIM Restrictions	1-40
System Code and MVSCP Data Separation	1-41
Using the Sample UIM	1-41

Chapter 2. Allocation Considerations 2-1

Serialization of Resources During Allocation	2-1
Improving Allocation Performance	2-2
The Device Preference Table	2-3
The Eligible Device Table	2-5
The Use of Esoteric Names	2-5
Creating Multiple EDTs	2-6
The Volume Attribute List	2-7
Use and Mount Attributes	2-7
Recovery of Allocated Resources	2-12
Controlling GRS Requests in MVS/XA	2-12

Modifying the System to Fit your Applications

Chapter 3. Limiting User Region Size	3-1
Setting a Default Region Size via JCL	3-1
Setting Default GETMAIN Limits via Exit Routines	3-1
The IEALIMIT Exit	3-2
The IEFUSI Exit	3-2
IEALIMIT Processing	3-3
The IEFUSI Interface	3-4
How VSM Uses the Region Size Value and the Limit Value	3-9
Chapter 4. Assigning Special Program Properties to Applications	4-1
Program Properties Table	4-1
Format of the PPT Table Header	4-2
Format of the PPT Entry	4-2
Contents of the PPT Entry	4-3
Updating the PPT	4-9
Chapter 5. Creating Your Own Resource Managers	5-1
Installation-Written Resource Managers	5-1
The Resource Manager Parameter List	5-2
Adding an Installation-Written Resource Manager	5-4
Chapter 6. Executing DAT-off Code in MVS/XA	6-1
Chapter 7. Controlling System Messages and the System Log	7-1
Controlling System Messages	7-2
Using a Message-Routing/Processing Exit Routine	7-3
Inserting A WTO/WTOR Exit Routine into the Control Program	7-5
Replacing a WTO/WTOR Exit Routine Without a re-IPL	7-7
The Hardcopy Log	7-7
Suppressing the Display of Selected Messages	7-8
Controlling the System Log	7-8
Modifying the System Log	7-9
Chapter 8. Updating the Master Job Control Language Data Set	8-1
IEEMSJCL Example	8-1
Changes to Master JCL	8-2
Alternate Versions of the Master JCL	8-2
Chapter 9. Customizing the System Trace Table	9-1
The USRn System Trace Table Entry	9-1

Using the PTRACE Macro	9-2
PTRACE Macro Processing	9-3
Formatting a USRn Trace Table Entry	9-4
Replacing a USRn TTE Formatting Routine	9-5

Fitting your Subsystems into the System

Chapter 10. Defining Subsystems To the Operating System	10-1
Defining Subsystems in Members of SYS1.PARMLIB	10-2
Passing Parameters to the Initialization Routine	10-3
System Handling of Duplicate Subsystems	10-4
Chapter 11. The Subsystem Affinity Service	11-1
SSAFF: Set/Obtain Subsystem Affinity	11-2
IEFSSREQ: Obtaining The SSAT Index Value	11-4

MVS/XA System Services

Chapter 12. Unit Verification Service	12-1
Unit Verification Service	12-1
Callers of IEFEB4UV	12-4
Callers of IEFGB4UV or IEFAB4UV	12-4
Input To and Output From Unit Verification Service Routines	12-5
Input and Output Data Structures	12-6
Requesting Multiple Functions	12-19
Example 1 Function Codes 0 and 1	12-19
Example 1 Function Codes 0 and 1 (continued)	12-20
Example 2 Function Codes 3 and 10	12-21
Example 3 Function Codes 1 and 5	12-22
Chapter 13. The Hot I/O Detection Table	13-1
IOSRHIDT: The HIDT	13-1
Modifying the HIDT	13-3
Chapter 14. The Internal Reader Facility	14-1
Setting Up and Using an Internal Reader	14-1
Requesting a Started Task To Execute on a Secondary Subsystem	14-6
Restrictions when Routing the JCL to the Master Subsystem	14-6
Defaults For The Subsystem	14-7
Chapter 15. The External Writer	15-1
STDWTR: IBM Standard Output Writing Routine	15-2
Writing Your Own Output Writing Routine	15-3
IEFSD094: The Output Separator Routine	15-8
The External Writer Cataloged Procedure	15-12
Chapter 16. The Virtual Fetch Service	16-1
Functions of Virtual Fetch	16-1
Installation Support for the Virtual Fetch Service	16-6
Starting Virtual Fetch	16-7
Refreshing Virtual Fetch	16-8
Considerations When Using Virtual Fetch	16-9
Programming Conventions for Using Virtual Fetch	16-11

Requesting Dumps When Using Virtual Fetch	16-11
BUILD Request for Virtual Fetch	16-13
FIND Request for Virtual Fetch	16-13
GET Request for Virtual Fetch	16-14
Chapter 17. MVS Dumping Services	17-1
MVS Dumps	17-2
ABEND Dumps	17-3
SVC Dumps	17-4
Suppressing Dumps	17-6
Suppressing Dumps Automatically, by Abend Code	17-6
Tailoring ABEND and SVC Dumps: The Dump Options	17-7
Tailoring Summaries and Symptom Dumps	17-8
Summary Dumps	17-9
Symptom Dumps	17-12
Tailoring Dumps: Other Data Options	17-13
Tailoring Dumps by Type: The Operator Commands	17-17
CHNGDUMP Operator Command	17-18
DISPLAY DUMP Operator Command	17-19
DUMPDS Operator Command	17-20
DUMP Operator Command	17-21
Tailoring and Suppressing Individual Dumps: The User Exits	17-21
Pre-Dump Exits for User Dumps	17-22
Post-Dump Exits for System Dumps	17-23
Dump Analysis and Elimination (DAE)	17-26
Definitions	17-26
Symptoms	17-27
Symptom Strings	17-27
Symptom Queue	17-27
Keys and Keywords	17-27
Minimum Symptoms	17-28
Input to DAE	17-29
DAE Parameter Record in SYS1.PARMLIB	17-29
ADYDFLT	17-30
SYS1.DAE	17-32
ABDUMP Symptom Area of the Dump Header Record	17-34
SDWA	17-34
DAE Processing	17-34
DAE Initialization	17-35
Symptom Extraction	17-35
How DAE Creates Symptoms	17-35
Criteria for DAE to Match for Duplicates	17-38
Criteria for DAE to Suppress Dumps	17-38
SUPPRESS and UPDATE Processing	17-39
Overrides to DAE	17-40
Creating and Modifying Symptom Data	17-40
VRA Keys for DAE	17-41
Adding to the Minimum Symptom String Requirements	17-42
Appendix A. IBM Provided Device Preference Table	A-1
Index	X-1

Figures

- 2-1. Processing Order Allocation Requests Requiring Serialization 2-2
- 2-2. Relationships among Generic and Esoteric Device Groups 2-6
- 2-3. Summary of Mount and Use Attribute Combinations 2-10
- 2-4. Sharable and Nonsharable Volume Requests 2-11
- 3-1. How VSM Arrives at Region Size and Limit Values from Values Set by IEALIMIT and IEFUSI. 3-5
- 3-2. Parameters Passed to Exit at IEFUSI by SMF 3-6
- 3-3. Effect of Region Size and Limit Values on Various GETMAIN Requests 3-9
- 5-1. Some Key Fields in the Resource Manager Parameter List (RMPL) 5-3
- 6-1. Using the DATOFF Macro to Execute DAT-off Code 6-2
- 8-1. IEEMSJCL Data Set 8-1
- 9-1. Examples of the PTRACE Macro 9-3
- 9-2. Continuation Information from PTRACE for Multi-Part TTE 9-4
- 9-3. Sample Code for Formatting USRn Trace Table Entries 9-8
- 10-1. Format of the SYS1.PARMLIB member, IEFSSNxx. 10-2
- 10-2. Parameter List for Subsystem Initialization Routines 10-3
- 11-1. Subsystem Affinity Service 11-1
- 12-1. Input Parameter List 12-5
- 12-2. Requesting Function Code 0 (Check Groups) 12-7
- 12-3. Requesting Function Code 1 (Check Units) 12-8
- 12-4. Requesting Function Code 2 (Return Unit Name) 12-9
- 12-5. Output from Function Code 2 (Return Unit Name) 12-9
- 12-6. Requesting Function Code 3 (Return UCB Addresses) 12-10
- 12-7. Output from Function Code 3 (Return UCB Addresses) 12-10
- 12-8. Requesting Function Code 4 (Return Group ID) 12-11
- 12-9. Output from Function Code 4 (Return Group ID) 12-11
- 12-10. Requesting Function Code 5 (Indicate Unit Name is a Look-up Value) 12-12
- 12-11. Requesting Function Code 6 (Return Look-up Value) 12-13
- 12-12. Output from Function Code 6 (Return Look-up Value) 12-13
- 12-13. Requesting Function Code 7 (Convert Device Type to Look-up Value) 12-14
- 12-14. Output from Function Code 7 (Convert Device Type to Look-up Value) 12-14
- 12-15. Requesting Function Code 8 (Return Attributes) 12-15
- 12-16. Requesting Function Code 10 (Specify Subpool for Returned Storage) 12-17
- 12-17. Requesting Function Code 11 (Return Unit Names for a Device Class) 12-18
- 12-18. Output from Function Code 11 (Return Unit Names for a Device Class) 12-18
- 12-19. Input for Function Codes 0 and 1 12-19

12-20.	Output from Function Codes 0 and 1	12-20
12-21.	Input for Function Codes 3 and 10	12-21
12-22.	Output from Function Codes 3 and 10	12-21
12-23.	Input for Function Codes 1 and 5	12-22
12-24.	Output from Function Codes 1 and 5	12-22
13-1.	Valid Hot I/O Recovery Action Codes	13-2
13-2.	IBM Default Hot I/O Threshold and Recovery Actions.	13-3
15-1.	External Writer Parameter List	15-3
15-2.	General Logic of IBM's External Writer Routine	15-5
15-3.	Parameter List for Separator Routine	15-9
16-1.	Environment Prior to Virtual Fetch Initialization	16-4
16-2.	Environment After Virtual Fetch Initialization	16-5
16-3.	Virtual Fetch Parameter List	16-10
16-4.	Environment After a BUILD/FIND Request	16-12
16-5.	Environment After a GET Request	16-16
16-6.	A Program Using Virtual Fetch	16-17
17-1.	Default Dump Options for ABEND and SVC Dumps	17-8
17-2.	ABEND Summary Dump Contents	17-9
17-3.	SVC Dump Summary Dump Contents	17-11
17-4.	Message IEA995I: Symptom Dump Output for SYSABEND and SYSUDUMP	17-12
17-5.	SDATA Options for MVS/XA Dumps	17-15
17-6.	Format of DAE Parameter Record	17-30
17-7.	Required Symptom Keys in ADYDFLT	17-31
17-8.	Optional Symptom Keys in ADYDFLT	17-31
17-9.	Sample JCL in SYS1.SAMPLIB for Creating SYS1.DAE	17-33
17-10.	Duplicate Areas in ABDUMP Symptom Area and SDWA	17-37

Summary of Amendments

Summary of Amendments for GC28-1152-2 for MVS/System Products Version 2 Release 2.0

This edition contains changes to support MVS/System Product Version 2 Release 2.0. This edition includes the following changes:

- A new chapter on Writing Unit Information Modules in support of the MVS Configuration Program (MVSCP) is included.
- Information on the MVS configuration program (MVSCP)
- Information on the Eligible Device Table Verification Service is no longer in this book; this information now appears in *MVS/Extended Architecture: MVS Configuration Program Guide and Reference*.
- The chapter on Assigning Special Program Properties to Applications is rewritten to support the SCHEDxx member of SYS1.PARMLIB.
- The chapter on Updating the Master Job Control Language Data Set is rewritten to support this release.
- The chapter on Defining Subsystems To the Operation System is rewritten to document support of the IEFSSNxx member of SYS1.PARMLIB.
- The chapter on Unit Verification Service is rewritten to include new functions and editorial changes.
- The chapter on MVS Dumping Services is rewritten to include new functions for IPCS and editorial changes.
- A new appendix is added containing the IBM provided device preference table values.

**Summary of Amendments
for GC28-1152-1
As Updated August 2, 1985
by Technical Newsletter GN28-1106**

This Technical Newsletter, which supports MVS/System Products Version 2 Release 1.3 Vector Facility Enhancement, contains the following updates:

- Information on processor affinity and the PPT in Chapter 3.
- Technical and editorial changes are included for maintenance.

**Summary of Amendments
for GC28-1152-1
As Updated January 30, 1984
by TNL GN28-0915**

This technical newsletter contains new and updated information in support of MVS/System Products Version 2 Release 1.2 and includes the following:

- New flag bits in the flag word of the VSM parameter list in Chapter 2, concerning the IEALIMIT and IEFUSI installation exit routines.
- New general-purpose WTO/WTOR user exit (IEAVMXIT) and user-specified WTO/WTOR exits in Chapter 6.
- New routing of messages according to the message level specified by the LEVEL keyword of the CONTROL V command, in Chapter 6.
- Minor technical and editorial changes, marked throughout the text by a change bar in the left margin.

Modifying the System to Fit the Devices in Your Installation

An MVS installation can include many input/output devices. They are among the *resources* available through the operating system that help accomplish the work of a data processing installation.

A system programmer is responsible for defining the I/O configuration to MVS and for coordinating the working of the devices with the operating system and the installation's applications.

This part of *System Modifications* presents a general description of defining the I/O configuration, and how the operating system allocates resources to do the work of the installation. It also includes descriptions of several means of influencing the allocation process.

The following books are referenced in this chapter:

- *MVS/XA MVS Configuration Program Guide and Reference*
- *MVS/XA Input/Output Configuration Program User's Guide and Reference*
- *MVS Planning: Global Resource Serialization*
- *MVS/XA JCL Reference*
- *MVS/XA Operations: System Commands*
- *MVS/XA SPL: Initialization and Tuning*
- *MVS/XA SPL: JES3 Initialization and Tuning*
- *MVS/XA SPL: System Generation Reference*
- *MVS/XA SPL: System Macros and Facilities, Volume 2*

Chapter 1. Writing Unit Information Modules for the MVS Configuration Program

Installations that install MVS/System Product Version 2 Release 2 (MVS/SP 2.2.0), or subsequent releases must use the MVS configuration program (MVSCP) to define the I/O configuration to MVS. For each supported device, MVSCP provides a program called a unit information module (UIM). It is the UIM that recognizes and processes the values coded on the IODEVICE statement which is part of the input stream to MVSCP. *MVS/Extended Architecture: MVS Configuration Program Guide and Reference* identifies the devices that MVSCP supports. A UIM may define the support for several related devices.

For devices that MVSCP does not support, you must supply the information in one of two ways: either define the device as a DUMMY device on the IODEVICE statement or write your own UIM to define the device to MVS. A device specified as DUMMY has some limitations. For example, the system creates a UCB for a unit record device. Because MVSCP does not support the ERRTAB or DEVTYPE parameter on the IODEVICE statement, the IBM-supplied default values may not fit your needs. The UCB will contain the default values for the ERP index, which is zero, and for the UCB type, which is X'00000800'. Because of these limitations, you may prefer to write a UIM instead of defining an unsupported device as a DUMMY. This chapter discusses writing UIMs for MVSCP.

Before reading this chapter, you should be familiar with *MVS/XA: MVS Configuration Program Guide and Reference*. While coding a UIM, you may need to use the *MVS/XA Debugging Handbooks* for relevant data areas.

This chapter discusses the following:

- How MVSCP uses UIMs
- UIM Data Areas
- UIM Service Routines
- UIM Processing Logic
- UIM Macros
- Device Support Modules and Macros
- Writing a UIM
- Using the Sample UIM

How MVSCP Uses UIMs

During its initialization, MVSCP loads all unit information modules into virtual storage. MVSCP makes three types of calls to the UIMs while building the I/O configuration: for initialization, to process the IODEVICE statements, and for end-of-data processing.

Initialization Processing

MVSCP communicates with each UIM through the UIM communication area (UCA). The UCA contains fields that a UIM can update and others that a UIM can read. During initialization, MVSCP calls each UIM to obtain the allocation information on all the generic device types recognized by a UIM. The UIMs call a service routine in the MVSCP to build the device information tables (DITs). The DIT contains information used to build the eligible device table (EDT). There is one DIT for each generic device type supported by the collection of UIMs. Each UIM also indicates if it must be invoked for end-of-data processing.

IODEVICE Statement Processing

For each IODEVICE statement, MVSCP invokes the UIMs until one of the UIMs recognizes the specified device. If no match is found, MVSCP issues an error message. If a match is found, the UIM validity checks the parameters on the IODEVICE statement. If they are valid, the UIM calls a service routine in MVSCP to build a device features table (DFT). The DFT contains information used to build the unit control blocks (UCBs) and the EDT. For each device number defined in the I/O configuration, there is a DFT.

End-of-Data Processing

MVSCP performs end-of-data processing for each UIM that, during initialization, indicated an end-of-data processing. During end-of-data processing, the UIM checks the I/O configuration for consistency and may update certain device dependent information.

UIM Data Areas

There are three control blocks, external to the UIM, that a UIM must reference:

- UIM communications area (UCA),
- Configuration program vector table (CPVT),
- IODEVICE internal text record (IODV).

See *MVS/XA Debugging Handbook* for mappings of these data areas.

The other data areas and parameter lists that a UIM uses are contained within the UIM itself.

UIM Communications Area (UCA)

The UCA contains information that MVSCP uses to communicate with the UIM. The UCA points to the CPVT, the DFT build routine, the DIT build routine, the getmain service routine, the feature checker routine, parameter checker routine, the device characteristics table build routine, and many other vital data areas. CBPZUCA maps the UCA.

MVSCP Vector Table (CPVT)

The CPVT is the MVSCP vector table. It points to the MVSCP service routines; it also contains anchors for global data structures and information concerning MVSCP. CBPZCPVT maps the CPVT.

IODEVICE Internal Text Record (IODV)

The IODV maps the IODEVICE internal text record. The IODEVICE internal text record is the control block representation of an IODEVICE statement. It contains the parameters and features that were specified on the IODEVICE statement. CBPZIODV maps the IODV.

UIM Service Routines

The MVS configuration program includes service routines to assist the UIMs. The following table identifies each routine and what it is used for.

A description for each of these service routines is on the following pages.

Service Routine Name	Function of the Routine
CBPADIT	Builds device information tables (DITs). There is one DIT for each generic device type supported by the collection of UIMs.
CBPDDCT	Builds device characteristics table (DCT). There is one DCT per I/O configuration. There is a separate entry in the DCT for each DASD type defined in the I/O configuration.
CBPIDFT	Builds the device features table (DFTs). There is one DFT per device in the I/O configuration.
CBPIFEAT	Checks the IODEVICE features.
CBPIGETM	Obtains storage for the UIM.
CBPIPARM	Validity checks the IODEVICE parameters.

CBPADIT - Builds DITs Service Routine

Invoking CBPADIT

A UIM must call CBPADIT once for each generic device that it defines. A separate DIT is built for each generic. All of the DITs are built before the MVSCP input stream is processed.

UIMs invoke CBPADIT in 31-bit addressing mode by using a BALR instruction. Use the standard register save area conventions. The address of the CBPADIT routine is in the field UCADITP in the UCA.

Registers on Entry to CBPADIT

Register 0	Undefined
Register 1	Pointer to a two word parameter list Word 1 - Address of the UCA Word 2 - Address of the DIP
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	CBPADIT entry point address

Registers on Exit from CBPADIT

Register 0-15	Restored
---------------	----------

CBPADIT Input Parameters: A UIM provides the input to CBPADIT in the device information parameters (DIP). The DIP resides in a UIM and is mapped by CBPYDIP.

CBPDDCT - Build DCT Service Routine

Invoking CBPDDCT

The UIMs invoke the CBPDDCT service routine to build DCT entries. A DCT entry is built for each type of DASD defined in the I/O configuration. A UIM must call CBPDDCT once for each DCT entry. IHADVCT, which is part of the Data Facilities Product, maps the DCT.

UIMs invoke CBPDDCT, in 31-bit addressing mode, by using a BALR instruction. Use standard register save area conventions. The UCADCTP field in the UCA contains the address of CBPDDCT.

Registers on Entry to CBPDDCT

Register 0	Undefined
Register 1	Pointer to a two word parameter list Word 1 - Address of the UCA Word 2 - Address of the DCP
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	CBPDDCT entry point address

Registers on Exit from CBPDDCT

Register 0-15	Restored
---------------	----------

CBPDDCT Input Parameters: A UIM provides the input to CBPDDCT in the device characteristics parameters (DCP). The DCP resides in a UIM and is mapped by CBPZDCP.

CBPIDFT - Build the DFTs Service Routine

Invoking CBPIDFT

UIMs invoke CBPIDFT to build device features tables (DFTs). DFTs contain the information that is used by the MVS configuration program to build UCBs. There is one DFT for each unit defined in the I/O configuration. One UCB is built from each DFT.

A UIM invokes CBPIDFT only after verifying that an IODEVICE statement contains no errors. A UIM must invoke CBPIDFT once for each DFT that is to be built. A DFT must be built for each device number defined by the IODEVICE statement. (For multiple exposure devices, a separate DFT must be built for each exposure.)

CBPIDFT is responsible for validating the device number that is to be assigned to the DFT. CBPIDFT ensures that the device number does not exceed X'FFF' and that it is unique within the the I/O configuration. If the device number is in error, CBPIDFT issues an appropriate error message, sets the IODVUINV flag in the IODV and returns to the UIM without building a DFT.

To invoke CBPIDFT within a UIM, use a BALR instruction in 31-bit addressing mode. Use standard register save area conventions. The UCADFTP field in the UCA contains the address of the CBPIDFT routine.

Registers on Entry to CBPIDFT

Register 0	Undefined
Register 1	Pointer to a two word parameter list Word 1 - Address of the UCA Word 2 - Address of the DFP
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	CBPIDFT entry point address

Registers on Exit from CBPIDFT

Register 0-15	Restored
---------------	----------

CBPIDFT Input Parameters: A UIM provides the input to CBPIDFT in the device features parameters (DFP). The DFP resides in a UIM and is mapped by CBPZDFP.

CBPIFEAT - Device Features Checker Service Routine

Invoking CBPIFEAT

UIMs invoke CBPIFEAT to determine:

1. Which, if any, of the features that are valid for the device have been specified on the IODEVICE statement.
2. If any features that are not valid for the device have been specified on the IODEVICE statement.

CBPIFEAT will issue an error message for each invalid feature that was specified and set the IODVUINV flag in the IODV.

UIMs invoke CBPIFEAT, in 31-bit addressing mode, by using a BALR instruction. Use standard register save area conventions. The field UCAFEATP in the UCA contains the address of the CBPIFEAT routine.

Registers on Entry to CBPIFEAT

Register 0	Undefined
Register 1	Pointer to a two word parameter list Word 1 - Address of the UCA Word 2 - Address of the FCP
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	CBPIFEAT entry point address

Registers on Exit from CBPIFEAT

Register 0-15	Restored
---------------	----------

CBPIFEAT Input Parameters: A UIM provides the input to CBPIFEAT in the features checker parameters (FCP). The FCP resides in a UIM and is mapped by CBPZFCP.

Note: The features that are valid for a device must have names that are from 1 to 10 characters long. The name given a particular feature is completely under the control of the UIM and is specified in the FCP.

CBPIGETM - Getmain Service Routine

Invoking CBPIGETM

UIMs invoke the CBPIGETM service routine to obtain a specified amount of storage to be used by the UIM as a work area. CBPIGETM zeros out the area before returning to the UIM.

UIMs invoke CBPIGETM, in 31-bit addressing mode, by using a BALR instruction. Standard register save area conventions are used. The field UCAGETMP in the UCA contains the address of CBPIGETM.

Registers on Entry to CBPIGETM

Register 0	Undefined
Register 1	Pointer to a two word parameter list Word 1 - Address of the UCA Word 2 - Address of the GETM
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	CBPIGETM entry point address

Registers on Exit from CBPIGETM

Register 0-15	Restored
---------------	----------

CBPIGETM Input Parameters: A UIM provides the input to CBPIGETM in the getmain parameters (GETM). The GETM resides in a UIM and is mapped by CBPZGETM.

CBPIPARM - IODEVICE Parameter Checker Service Routine

Invoking CBPIPARM

UIMs invoke CBPIPARM to determine if:

1. Any required parameters for the particular device were not specified on the IODEVICE statement.

CBPIPARM will issue an error message for each missing parameter and set the IODVUINV flag in the IODV.

2. Any parameters were specified on the IODEVICE statement are inappropriate for the particular device.

CBPIPARM issues an informational message for each inappropriate parameter that was specified.

UIMs invoke CBPIPARM, in 31-bit addressing mode, by using a BALR instruction. Use standard register save area conventions. The field UCAPARMP in the UCA contains the address of CBPIPARM.

Registers on Entry to CBPIPARM

Register 0	Undefined
Register 1	Pointer to a two word parameter list Word 1 - Address of the UCA Word 2 - Address of the PCP
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	CBPIPARM entry point address

Registers on Exit from CBPIPARM

Register 0-15	Restored
---------------	----------

CBPIPARM Input Parameters: A UIM provides the input to CBPIPARM in the parameter checker parameters (PCP). The PCP resides in a UIM and is mapped by CBPZPCP.

UIM Processing Logic

During MVSCP's calls to the UIM, the UCAUIMRT field in the UCA indicates the type of call: initialization, IODEVICE statement checking, or end-of-data processing.

During the MVSCP's initialization call to the UIM, (UCAUIMRT = UCARINIT), the UIM does the following:

- Invokes the CBPZPPDS macro to add an entry to the diagnostic stack
- For each generic type defined by this UIM, the UIM:
 - Builds the CBPYDIP parameter list
 - Invokes the CBPADIT routine to build the DIT
- If the UIM needs a work area, it
 - Builds the CBPZGETM parameter list
 - Invokes CBPIGETM to get the work area
- If necessary, the UIM indicates in the UCA that it must be called for end-of-data processing.
- Invokes the CBPZPPDS macro to remove an entry from the diagnostic stack

During the MVSCP's IODEVICE statement checking call (UCAUIMRT = UCARDFTB) the UIM does the following:

- Invokes the CBPZPPDS macro to add an entry to the diagnostic stack
- If the unit in the IODV is one of the units defined by this UIM
 - Builds the CBPZPCP parameter list
 - Invokes CBPIPARM to check the IODEVICE parameters
 - If one or more features are supported:
 - Builds the CBPZFCP parameter list
 - Invokes CBPIFEAT to check the IODEVICE features.
 - Validity checks the IODEVICE internal text record
 - Invokes the CBPZLOG macro to issue messages, if necessary
 - If the IODEVICE internal text record is valid, then for each unit defined in the IODV:
 - Builds the CBPZDFP parameter list
 - Invokes CBPIDFT to build the DFT
 - If necessary, saves the following addresses for end-of-data processing:
 - UCADDSP
 - UCADDEP
 - UCADCEP
 - Indicates in the UCA that the IODEVICE internal record was processed
- Invokes the CBPZPPDS macro to remove an entry from the diagnostic stack

During MVSCP's end-of-data processing call (UCAUIMRT = UCAREOD), the UIM performing the end-of-data processing does the following:

- Invokes the CBPZPPDS macro to add an entry to the diagnostic stack
- Performs end-of-data checking
- Invokes the CBPZLOG macro to issue messages, if necessary
- Updates only for the devices defined by this UIM the following, if necessary:
 - UCB device dependent segment
 - UCB device dependent extension
 - UCB device class extension
- For each type of DASD supported by this UIM and defined in the I/O configuration, the UIM:
 - Builds the CBPZDCP parameter list
 - Invokes CBPDDCT routine to build the DCT entry
- Invokes the CBPZPPDS macro to remove an entry from the diagnostic stack

Considerations for UIM Processing

UIMs are invoked in task mode and in problem state. However, do not code a UIM to depend on being invoked in this manner. A UIM can not invoke any other module or system service except for those service routines provided by MVSCP.

UIMs are invoked in 31-bit addressing mode. UIMs must not change to 24-bit addressing mode. Link-edit UIMs with AMODE(31) and RMODE(ANY).

Each time the MVSCP is invoked, a fresh copy of each UIM is loaded into virtual storage. The same copy of each UIM is used throughout the processing of the invocation of MVSCP. This allows a UIM to store information within itself and to retain this information for subsequent calls to that UIM.

UIMs must use the standard register save area conventions. The UIM must set register 13 to point to its own register save area before invoking any UIM service routines or before using the CBPZPPDS or CBPZLOG macros.

Entry logic for a UIM

Upon entry for a UIM:

- Save the contents of the input registers
- Set the UIM base register
- Chain the save areas
- Set register 13 to point to the save area contained within the UIM
- Establish addressability to the UCA, CPVT and IODV
The IODV is present only on IODEVICE calls to the UIM.
- Issue the CBPZPPDS macro to put an entry on the diagnostic stack

Registers on Entry to a UIM

Upon entry to a UIM, the registers are defined as follows:

Register 0	Undefined
Register 1	Pointer to a fullword containing the UCA address
Register 2-12	Undefined
Register 13	Address of an 18-word save area
Register 14	Return address
Register 15	UIM entry point address

Exit logic from a UIM

Upon exit from a UIM:

- Issue the CBPZPPDS macro to remove the UIM's entry from the diagnostic stack
- Restore the caller's registers
- Return to the caller

Registers on Exit from a UIM

Upon exit from a UIM, the registers are defined as follows:

Register 0-15	Restored
---------------	----------

UIM Macros

The following macros are used by UIMs. These macros reside in SYS1.AMODGEN.

Macro Name	Function of the Macro
CBPYDIP	Maps the device information parameters (DIP) that provide input to CBPADIT.
CBPZCPVT	Maps the configuration program vector table (CPVT).
CBPZDCP	Maps the device characteristics parameters (DCP), that provides input to CBPDDCT.
CBPZDFP	Maps the device features parameters (DFP), that provides input to CBPIDFT.
CBPZDIAG	Builds an MVSCP diagnostic stack entry.
CBPZFCP	Maps the feature checker parameters (FCP), that provides input to CBPIFEAT.
CBPZGETM	Maps the getmain parameters (GETM) that provide input to CBPIGETM.
CBPZIODV	Maps the IODEVICE internal text record.
CBPZITRH	Maps the header of the IODEVICE internal text record.
CBPZLOG	Macro used to invoke the message log routine.
CBPZLOGR	Maps the input parameters to the message log routine.
CBPZPCP	Maps the parameter checker parameters (PCP), that provide input to CBPIPARAM.
CBPZPPDS	Macro used to push an entry on or pop an entry from the MVSCP diagnostic stack.
CBPZUCA	Maps the UIM communications area.

See *MVS/XA Debugging Handbooks* for the mapping macros.

This section contains information and syntax for the preceding UIM macros.

CBPZDIAG and CBPZPPDS Macros

These two executable macros are needed within a UIM to support the recovery routine in MVSCP.

A UIM must not establish its own recovery routine. Instead, the CBPZDIAG macro allows you to specify diagnostic information to MVSCP's recovery routine. Use the CBPZDIAG macro to build a diagnostic stack entry in which you specify certain diagnostic information. If an abend occurs, this information is placed in the system diagnostic work area (SDWA).

The CBPZPPDS macro puts an entry on (pushed on) or takes an entry off (popped off) the MVSCP diagnostic stack. Before issuing this macro, the UIM must have addressability to MVSCP's vector table (CPVT).

CBPZLOG and CBPZLOGR Macros

Use the CBPZLOG macro in your UIM to issue messages. Do not use message ids that IBM uses. Use message ids in the range CBP900I-CBP999I.

The CBPZLOG macro requires the CBPZLOGR macro. The CBPZLOGR macro maps the parameter list of the message log routine. This parameter list resides in the UIM and is built by the code generated in the CBPZLOG macro expansion. A UIM must have addressability to MVSCP's vector table (CPVT) when it issues the CBPZLOG macro.

See *MVS/XA Debugging Handbooks* for the data areas.

CBPYDIP

The CBPYDIP macro maps the device information parameters (DIP). The DIP is the input parameter list to CBPADIT.

The syntax of the CBPYDIP macro is as follows:

```
CBPYDIP [DENS=dens],  
        [GENDNMS=gendnms]
```

dens specifies the number of entries to be generated in the density list. This list contains the densities that are supported by the generic device type. This parameter is optional, the default is 0.

gendnms specifies the number of entries to be generated in the compatible generic device name list. This list contains the generic names of devices for which this generic device type can be used to satisfy allocation requests. This parameter is optional, the default is 0.

Note: You cannot specify a label on the CBPYDIP macro invocation.

CBPZCPVT

The CBPZCPVT macro maps the configuration program vector table (CPVT). The CPVT points to many of the internal control blocks and service routines used by the MVSCP. It also contains parameters used by some of these service routines.

A UIM never directly references the CPVT, but it must have addressability to the CPVT when it issues the CBPZPPDS or CBPZLOG macros.

The syntax of the CBPZCPVT macro is as follows:

```
CBPZCPVT [CSECT=csect]
```

csect this operand should *never* be specified by a UIM.

Note: You cannot specify a label on the CBPZCPVT macro invocation.

CBPZDCP

The CBPZDCP macro maps the device characteristics parameters (DCP). The DCP is the input parameter list to CBPDDCT.

The syntax of the CBPZDCP macro is as follows:

```
CBPZDCP
```

There are no input parameters on the CBPZDCP macro invocation.

Note: You cannot specify a label on the CBPZDCP macro invocation.

CBPZDFP

The CBPZDFP macro maps the device features parameters (DFP). The DFP is the input parameter list to CBPIDFT.

The syntax of the CBPZDFP macro is as follows:

```
CBPZDFP  [MLTS=mlts],  
         [RELOC=reloc]
```

- mlts** specifies the number of entries to be generated in the MLT list. This list identifies the MLT(s) that designate the nucleus and LPA device support modules for the device. This parameter is optional, the default is 1. (The maximum number of MLTs allowed for a device is 5.)
- reloc** specifies the number of entries to be generated in the relocation list. The relocation list identifies fields in the device dependent sections of the UCB (device dependent segment, device dependent extension or device class extension) that point to other sections of the same UCB or another UCB. This parameter is optional, the default is 0.

Note: You cannot specify a label on the CBPZDFP macro invocation.

A UIM may not specify more than 256 bytes of certain device dependent information for a device. The information that falls within this 256 byte limit consists of:

- UCB device dependent segment (length specified in DFP field DFPDDSL)
- UCB device dependent extension (length specified in DFP field DFPDDEL)
- UCB device class extension (length specified in DFP field DFPDCEL)
- MLT list (the length of the list is computed by multiplying the number of entries in the list, which is contained in DFP field DFPMLTLC, by the length of a list entry, which is 12 bytes)
- Relocation list (the length of the list is computed by multiplying the number of entries in the list, which is contained in DFP field DFPRELCT, by the length of a list entry, which is 12 bytes)

CBPZDIAG

The CBPZDIAG macro is used to build a diagnostic stack entry. The diagnostic stack entry contains debugging information that is placed in the system diagnostic work area (SDWA) if an ABEND occurs in the UIM. The diagnostic stack entry is contained within the UIM.

A UIM must not establish an ESTAE to provide diagnostic information in the event that it abends. Rather, it must:

1. Specify the diagnostic information in a diagnostic stack entry, using the CBPZDIAG macro.
2. Use the CBPZPPDS macro to put the entry on the diagnostic stack in its entry logic.
3. Use the CBPZPPDS macro to remove the entry from the diagnostic stack in its exit logic.

The MVSCP ESTAE routine uses the information in the active diagnostic stack entry to fill in the SDWA. Also, the ESTAE routine builds a symptom string in the variable recording area (VRA) consisting of all the CSECT names in the entries on the diagnostic stack.

The syntax of the CBPZDIAG macro is as follows:

```
label      CBPZDIAG MODNAME=modname,  
           [CSECT=csect],  
           COMP=comp,  
           DESC=desc,  
           [VRADATA=vradata],  
           [RELATED=('related')]
```

label name of the diagnostic stack entry. The labels on the fields generated in the diagnostic stack entry will start with the same characters as **label** does. (In the event that **label** exceeds four characters, only the first four characters will be used in building the labels on the generated fields.) **label** is required.

modname name of the load module that contains the diagnostic stack entry. If an ABEND occurs, this value will be placed in SDWA field SDWAMODN. This parameter is required.

csect name of the CSECT that contains the diagnostic stack entry. If an ABEND occurs, this value will be placed in SDWA field SDWAC SCT. This parameter is optional, the default is the assembler symbol &SYSECT value.

comp component identifier of the UIM. If an ABEND occurs, this value will be placed in SDWA field SDWACID. The component identifier should be five bytes long. This parameter is required.

desc UIM description, which should contain the unit names of the device(s) that the UIM supports. If an ABEND occurs, this value will be placed in SDWA field SDWASC. The UIM description can be a maximum of 23 bytes long. This parameter is required.

vradata name of an array that contains the addresses of data to be placed in the VRA, if an ABEND occurs. The array contains the VRA keys and data lengths, in addition to the data addresses. This parameter is optional. If it is not specified, no specific control blocks or data areas for the UIM will be placed in the VRA. (On IODEVICE calls, the diagnostic stack entry for CBPICBBR, which is the routine that invokes UIMs on IODEVICE calls, causes the IODV to be placed in the VRA.)

Each entry in the VRA array contains eight bytes. The format of an entry is as follows:

Offset	Length	Function
0	2	Reserved, must be set to zero in all but the last entry in the array.
2	1	Key of VRA data, as specified in IHAVRA.
3	1	Length of VRA data.
4	4	Address of VRA data. If this field is set to zero, the ESTAE routine will skip this entry when moving data into the VRA. UIMs are permitted to dynamically update this field while the diagnostic entry is on the diagnostic stack.

The last entry in the VRA array must be set to X'FFFFFFFFFFFFFFFF'. This entry denotes the end of the VRA array and does not cause any data to be placed in the VRA.

related optional character string.

CBPZFCP

The CBPZFCP macro maps the feature checker parameters (FCP). The FCP is the input parameter list to CBPIFEAT. The FCP lists the features that are supported for a device.

The syntax of the CBPZFCP macro is as follows:

```
CBPZFCP  [FEAT=feat],  
         [CFEAT=cfeat]
```

feat specifies the number of entries to be generated in the feature list. This list identifies the features that are supported for a device. A maximum of 64 features can be supported for a device. This parameter is optional, the default is 0.

cfeat specifies the number of entries to be generated in the compatible feature list. This list identifies obsolete features for the device that are still permitted to be specified on the IODEVICE statement for compatibility considerations. This parameter is optional, the default is 0.

Note: You cannot specify a label on the CBPZFCP macro invocation.

CBPIFEAT identifies which features, if any, in the feature list were specified on the IODEVICE statement by setting the appropriate bit in the FCPOUT field. (The bits in the FCPOUT field map one-to-one with the entries in the feature list.)

CBPIFEAT issues an error message for each feature specified on the IODEVICE statement that is not in either the feature list or the compatible feature list. CBPIFEAT issues an informational message for each feature specified on the IODEVICE statement that is in the compatible feature list. (CBPIFEAT does not notify the UIM as to which features in the compatible feature list were specified on the IODEVICE statement.) CBPIFEAT also issues an error message for each feature specified more than once on the IODEVICE statement.

If CBPIFEAT issues an error message, it also sets the IODVUINV flag in the IODV to indicate that the IODEVICE internal text record is invalid.

CBPZGETM

The CBPZGETM macro maps the getmain parameters (GETM). The GETM is the input parameter list to CBPIGETM.

The syntax of the CBPZGETM macro is as follows:

```
CBPZGETM
```

There are no input parameters on the CBPZGETM macro invocation.

Note: You cannot specify a label on the CBPZGETM macro invocation.

CBPZIODV

The CBPZIODV macro maps the IODEVICE internal text record (IODV).

The syntax of the CBPZIODV macro is as follows:

```
CBPZIODV
```

There are no input parameters on the CBPZIODV macro invocation.

Note: You cannot specify a label on the CBPZIODV macro invocation.

A UIM should never invoke the assembler form of CBPZIODV. It is invoked automatically by the CBPZITRH macro, which maps the header section of internal text records. (The CBPZIODV macro will not assemble without the CBPZITRH macro.)

The IODV is basically a read-only control block. The only field in the IODV that a UIM is permitted to set is the IODVUINV flag. (The IODVUINV flag is set when an error is detected in the IODV.)

CBPZITRH

The CBPZITRH macro maps the internal text record headers (ITRH). (An internal text record is the control block representation of an MVSCP input statement.)

The assembler form of the CBPZITRH macro invokes the CBPZIODV macro. The CBPZITRH and CBPZIODV macros combine to give a complete mapping of the IODEVICE internal text record. (The CBPZITRH macro is required for the CBPZIODV macro to assemble.)

The only field in the ITRH that a UIM should ever reference is ITRHSNBR, which contains the number of the associated input statement. A UIM must *never* modify any of the ITRH fields.

The syntax of the CBPZITRH macro is as follows:

```
CBPZITRH
```

There are no input parameters on the CBPZITRH macro invocation.

Note: You cannot specify a label on the CBPZITRH macro invocation.

CBPZLOG

The CBPZLOG macro is used to issue a message to the MVSCP message log file. A UIM must have addressability to the CPVT when it issues the CBPZLOG macro. It must also invoke the CBPZLOGR mapping macro. (CBPZLOGR maps the parameter list that is built by the CBPZLOG macro.)

The syntax of the CBPZLOG macro is as follows:

```
label      CBPZLOG  MID=mid,
              [SEV=sev],
              [STMT=stmt],
              TEXT=text
```

label name of the label to be generated on the first instruction in the macro expansion. **label** is optional.

mid message identifier. The message identifier is seven characters long and is in the form of CBPnnnI, where nnn is a decimal number from 900 to 999 inclusive for customer-written UIMs. This parameter is required.

sev message severity. The following severities are supported:

LOGRINFO informational message. This message has no effect on MVSCP processing or its return code.

LOGRWARN warning message. This message has no effect on MVSCP processing but will cause a return code of 4 to be issued (unless a higher severity message is issued.)

LOGRERR error message. This message will prevent the MVSCP from building any I/O configuration members, and will cause a return code of 8 to be issued (unless a higher severity message is issued.)

LOGRTERM terminating message. This message causes the MVSCP to terminate its processing and issue a return code of 16. A UIM must *never* issue a terminating message.

This parameter is optional, the default is LOGRERR.

Note: The equates LOGRINFO, LOGRWARN, LOGRERR and LOGRTERM are generated by the CBPZLOGR macro.

stmt number of the statement in the MVSCP input stream that the message refers to. Field ITRHSNBR in the internal text record header (mapped by CBPZITRH) contains the statement number. This parameter is optional. If it is omitted, no statement number will be associated with the message.

text message text. This field contains up to 255 bytes of message text. The length of the text is determined by the length attribute of this field. This parameter is required.

Note: The message service will compress multiple blanks in the text and will split the text across multiple lines if necessary.

CBPZLOGR

The CBPZLOGR macro maps the message log routine's input parameter list. The parameter list is built by the CBPZLOG macro, which invokes CBPMLOGR.

The syntax of the CBPZLOGR macro is as follows:

```
CBPZLOGR
```

There are no input parameters on the CBPZLOGR macro invocation.

Note: You cannot specify a label on the CBPZLOGR macro invocation.

CBPZPCP

The CBPZPCP macro maps the parameter checker parameters (PCP). The PCP is the input parameter list to CBPIPARM. The PCP identifies (1) the parameters that are required for a device and (2) the parameters that are supported for a device (supported parameters consist of required parameters and optional parameters).

PCP field PCPREQD is used to indicate which IODEVICE parameters are required for a device, while field PCPSUPP is used to indicate which IODEVICE parameters are supported for a device. The IODVPRMS field (contained within the IODV) must be mapped over the PCPREQD and PCPSUPP fields in order to set the bits corresponding to the required and supported parameters.

The syntax of the CBPZPCP macro is as follows:

```
CBPZPCP
```

There are no input parameters on the CBPZPCP macro invocation.

Note: You cannot specify a label on the CBPZPCP macro invocation.

CBPIPARM issues an error message for each required parameter that is not specified on the IODEVICE statement. It also sets the IODVUINV flag in the IODV, when one or more required parameters are not specified, to indicate that the IODEVICE internal text record is invalid.

CBPIPARM issues an informational message for each unsupported parameter that is specified on the IODEVICE statement.

CBPZPPDS

The CBPZPPDS macro is used to push an entry on (put an entry on) or pop an entry from (remove an entry from) the diagnostic stack. A UIM must have addressability to the CPVT when it issues the CBPZPPDS macro. It must also have invoked the CBPZDIAG macro to build the diagnostic stack entry that is to be pushed on or popped from the diagnostic stack.

The syntax of the CBPZPPDS macro is as follows:

```
label      CBPZPPDS {PUSH | POP},  
           DIAG=diag  
           [,RELATED=('related')]
```

- label** name of the label to be generated on the first instruction in the macro expansion. **label** is optional.
- PUSH** The designated diagnostic entry is to be put on the diagnostic stack. Either PUSH or POP must be specified.
- POP** The designated diagnostic entry is to be removed from the diagnostic stack. Either PUSH or POP must be specified.
- diag** name of the diagnostic entry. This name must be specified on the label field of the CBPZDIAG macro invocation.
- related** optional character string.

CBPZUCA

The CBPZUCA macro maps the UIM communications area (UCA).

The syntax of the CBPZUCA macro is as follows:

CBPZUCA

There are no input parameters on the CBPZUCA macro invocation.

Note: You cannot specify a label on the CBPZUCA macro instruction.

The UCA contains some fields that are unrelated to the UIM processing. A UIM may only reference the (UCA) fields listed below. (The only fields in the list that a UIM may modify are UCARECOG and UCAEODAT.)

UCACPVTP points to CPVT.

UCADFTP points to DFT build routine (CBPIDFT).

UCADITP points to DIT build routine (CBPADIT).

UCAFEATP points to IODEVICE feature checker (CBPIFEAT).

UCAPARMP points to IODEVICE parameter checker (CBPIPARM).

UCAIODVP points to IODEVICE internal text record (IODV), when UCAUIMRT is set to UCARDFTB.

UCADDSF points to the UCB device dependent segment data. This value is set by CBPIDFT when a UIM has specified a device dependent segment in the DFP. This value points to the area within the DFT that will be used to build the UCB device dependent segment. If the UIM needs to modify the data in this area on its end-of-data call, it must save this address after calling CBPIDFT.

UCADDEP points to the UCB device dependent extension data. This value is set by CBPIDFT when a UIM has specified a device dependent extension in the DFP. This value points to the area within the DFT that will be used to build the UCB device dependent extension. If the UIM needs to modify the data in this area on its end-of-data call, it must save this address after calling CBPIDFT.

UCADCEP points to the UCB device class extension data. This value is set by CBPIDFT when a UIM has specified a device class extension in the DFP. This value points to the area within the DFT that will be used to build the UCB device class extension. If the UIM needs to modify the data in this area on its end-of-data call, it must save this address after calling CBPIDFT.

- UCAUIMRT** UIM request type. Set to one of the following values:
- UCARINIT** UIM is being called to perform its initialization function. None of the IODEVICE statements have been processed yet.
 - UCARDFTB** UIM is being called to process an IODEVICE internal text record (IODV).
 - UCAREOD** UIM is being called to perform its end-of-data function. All of the IODEVICE statements have been processed.
- UCARECOG** set by a UIM when it recognizes the unit parameter in the IODV. When a UIM sets UCARECOG, it is responsible for processing the IODV. (The UCARECOG flag can only be set on IODEVICE calls to a UIM.)
- UCAEODAT** set by a UIM when it wants an end-of-data call after all the IODVs have been processed. The UCAEODAT flag can only be set on the initialization call to a UIM. If a UIM does not set this flag on its initialization call, then it will not receive an end-of-data call.
- UCADCTP** points to the DCT build routine (CBPDDCT).
- UCAGETP** points to the getmain routine (CBPIGETM).

Device Support Modules and Macros

Any module that is required in the system if a particular device(s) is defined in the I/O configuration is a *device support module*. Each nucleus device support module must be a member of SYS1.NUCLEUS. Each LPA device support module must be a member of SYS1.LINKLIB or any data set within the link list concatenation.

Note: Nucleus device support modules that contain more than one control section must be link-edited with the scatter (SCTR) option.

If you write a nucleus or LPA device support module, you must provide a module lists table (MLT) containing the nucleus and the LPA device support modules for a given device. Use the IOSDMLT macro to build a MLT.

If you write your own device support module, you will have to provide a device descriptor table (DDT). Use the IOSDDT macro to build the DDT.

Before writing a device support module, or an error recovery procedure, read the following:

1. *MVS/XA SLL Input/Output Supervisor*
2. *IBM System/370 Extended Architecture Principles of Operations*
3. *IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers' Information*

IOSDMLT and IOSDDT Macros

Each device that is defined in an I/O configuration must have a module lists table (MLT) and a device descriptor table (DDT) associated with it. Similar devices can share the same DDT and MLT. Both tables must reside in SYS1.NUCLEUS. The UIM for a device must specify the name of the MLT and the DDT associated with that device. This is accomplished through the parameter list to CBPIDFT.

The IOSDMLT macro builds a module lists table (MLT). The MLT contains the names of the device support routines and, for each routine, it indicates whether the routine is located in the nucleus or in LPA.

The IOSDDT macro builds a device descriptor table (DDT). The DDT is the vector table to the device dependent exits for a device. The following pages describe these macros in more detail.

IOSDDT - Device Descriptor Table Build Macro

The IOSDDT macro builds a device descriptor table (DDT). The DDT, which must reside in SYS1.NUCLEUS, is the vector table to the device dependent exits for a device. The IOSDDT macro is located in SYS1.AMODGEN.

A device descriptor table (DDT) is a vector table that IOS uses to locate the device support routines. The system requires one of these tables for each device in the I/O configuration, although similar devices may share the same DDT. When conditions arise during I/O operations for which specific device dependent processing is required, IOS gives control to the exit routines through the vector entries in the DDT.

To build the DDT, you use the IOSDDT macro. With this macro, you specify either the entry point name or the module name of the DDT exit routines for the devices supported by that DDT. These exit routines perform the processing for various system functions that occur when the system performs I/O operations. The parameters of the IOSDDT macro allow you to specify the following kinds of routines, which receive control from IOS when the appropriate condition arises:

- The start I/O exit routine
- The trap exit routine
- The translate CCW table
- The ERP message routine
- The DDR exit routine
- The unsolicited interrupt exit routine
- The sense exit routine
- The end of sense exit routine
- The MIH exit routine
- The device service exit routine
- The channel program scan exit routine
- The subsystem ID

The information in the DDT is created from the parameters of the IOSDDT macro. The label that you specify on the IOSDDT macro is required because it is used as the CSECT name for the DDT being generated. When the system is IPLed, the DDT for each device in the I/O configuration becomes part of the nucleus. Each use of the IOSDDT macro generates one DDT.

The IOSDDT macro instruction is written as follows:

<i>name</i>	<i>name:</i>
b	One or more blanks must precede IOSDDT.
IOSDDT	
b	One or more blanks must follow IOSDDT.

SIOEXIT = <i>epname</i>	<i>entry point name</i>
,TRPEXIT = <i>epname</i>	<i>entry point name</i>
,TCCWTAB = <i>epname</i>	<i>entry point name</i>
,ERPEXIT = (<i>epname,type</i>)	<i>entry point name</i>
[,DDREXIT = (<i>epname,type</i>)	<i>entry point name</i>]
[,UNSEXIT = <i>epname</i>	<i>entry point name</i>]
[,SNSEXIT = <i>epname</i>	<i>entry point name</i>]
[,EOSEXIT = <i>epname</i>	<i>entry point name</i>]
[,MIHEXIT = <i>epname</i>	<i>entry point name</i>]
[,DSEXIT = <i>epname</i>	<i>entry point name</i>]
[,CPSEXIT = <i>epname</i>	<i>entry point name</i>]
[,SSYSID = <i>ssname</i>	<i>subsystem name</i>]

The parameters are explained as follows:

name

specifies name of the DDT. IOSDDT uses this name on the CSECT statement that it generates for the DDT. The name parameter is **required**.

SIOEXIT = *epname*

specifies the name of the start I/O exit entry point. This parameter is **required**.

TRPEXIT = *epname*

specifies the name of the trap exit entry point. This parameter is **required**.

TCCWTAB = *epname*

specifies the name of the translate CCW table entry point. This parameter is **required**.

ERPEXIT = (*epname,type*)

specifies the name of the ERP message entry point. Type describes whether the entry point name is to be treated as an entry point name address or a module name. Type can be specified as A for address or N for EBCDIC name. If A is specified, the entry point name will be resolved into an address. The module is loaded into the nucleus region from SYS1.NUCLEUS. If N is specified, The last 4 characters of the module name will be placed in the DDT. The module is loaded into the LPA from the LINK LIST concatenation. If neither is specified, N is the default. This parameter is **required**.

DDREXIT = (*epname,type*)

specifies the name of the DDR exit entry point. Type describes whether the entry point name is to be treated as an entry point name address or a module name. Type can be specified as A for address or N for EBCDIC name. If A is specified, the entry point name will be resolved into an address. The module is loaded into the nucleus region from SYS1.NUCLEUS. If N is specified, The last 4 characters of the module name will be placed in the DDT. The module is loaded into the LPA from the LINK LIST concatenation. If neither is specified, N is the default.

UNSEXIT = *epname*

specifies the name of the unsolicited interrupt exit entry point.

SNSEXIT = *epname*

specifies the name of the sense exit entry point.

EOSEXIT = *epname*

specifies the name of the end of sense exit entry point.

MIHEXIT = *epname*

specifies the name of the MIH exit entry point.

DSEXIT = *epname*

specifies the name of the device service exit entry point.

CPSEXIT = *epname*

specifies the name of the channel program scan exit entry point.

SSYSID = *ssname*

specifies the name of the subsystem ID, which can be one to four characters.

Note: When both ERPEXIT and DDREXIT are specified as EBCDIC module names, IOSDDT verifies that both specified module names have the same 4-character prefix. If the prefixes are not the same, IOSDDT issues an MNOTE and not does generate a DDT.

IOSDMLT - Module Lists Table Macro

The IOSDMLT macro builds a module lists table (MLT).

The module lists table (MLT) must reside in SYS1.NUCLEUS. It identifies the nucleus and LPA modules required to support the device you are defining, and that need to be loaded during the IPL process. For example, the MLT for an unsupported printer would designate all the modules that must be loaded into the nucleus and the LPA to support that printer. Note that the MLT must list all the nucleus and LPA device support modules for the device regardless of whether the modules are provided by you or by IBM.

To build a module lists table, use the IOSDMLT macro. Each IOSDMLT macro that you code creates an MLT CSECT. The label specified on the IOSDMLT macro, which is required, is used as the CSECT name. As parameters of the IOSDMLT macro, you specify a set of nucleus-resident module names and a set of LPA-resident module names. Each use of the IOSDMLT macro generates one MLT, which resides in a separate module. The IOSDMLT macro resides in SYS1.AMODGEN.

The IOSDMLT macro instruction is written as follows:

<i>name</i>	<i>name:</i>
b	One or more blanks must precede IOSDMLT.
IOSDMLT	
b	One or more blanks must follow IOSDMLT.

NUCL = (<i>nucid</i> < , <i>nucid</i> > ...)	<i>nucid</i> : name of nucleus module
,LPAL = (<i>lpaid</i> < , <i>lpaid</i> > ...)	<i>lpaid</i> : name of LPA module

The parameters are explained as follows:

name

specifies the name of the MLT. IOSDMLT uses this name on the CSECT statement that it generates for the MLT. The name parameter is required.

NUCL = (*nucid* < ,*nucid* > ...)

specifies the names of the nucleus modules that are to be loaded from SYS1.NUCLEUS into the nucleus region if the device associated with this MLT is defined in the I/O configuration.

,LPAL = (*lpaid* < ,*lpaid* > ...)

specifies the names of the LPA modules that are to be loaded from the LINK LIST concatenation into LPA if the device associated with this MLT is defined in the I/O configuration.

Note: IOSDMLT generates an END statement at the end of its expansion.

Writing a UIM

This section contains information on the naming conventions for UIMs, a partial list of IBM-supplied UIMs, and description of using the sample UIM that IBM supplies in SYS1.SAMPLIB.

Naming a UIM

IBM-supplied UIMs have member names of CBPUSxxx, where xxx is a decimal number from 001 to 256. User-supplied UIMs must have member names of CBPUCxxx, where xxx is a decimal number from 001 to 256.

UIMs must reside in SYS1.LINKLIB or the data set containing the UIM can be specified on the STEPLIB DD statement when invoking the MVSCP.

CAUTION: The system uses the members in the STEPLIB first. Therefore, if an installation-written UIM uses the name of an IBM-supplied UIM, the system will use the information in the installation-written UIM; not IBM's.

IBM-supplied UIMs

UIMs supplied by IBM are part of the product that supports the associated device. For example, the UIM supporting 3375s and 3380s is part of the Data Facility Product. Therefore, your installation has access to UIMs only for the products it uses. Some device types are defined as another device type. See *MVS/XA MVS Configuration Program Guide and Reference* to determine if a particular device is supported by MVS/XA.

Following is a *partial* list of the IBM-supplied UIMs, the product that contains the UIM and the devices the UIM defines. If you are creating a UIM to support a device that is similar to one that IBM-supplies, you can use the IBM-supplied UIM as an example. Be sure to name your UIM according to the requirements.

Devices	UIM Name	Product
1050 1050X 115A	CBPUS024	MVS
1287 1288	CBPUS032	MICR/OCR
1403	CBPUS012	DFP
2250	CBPUS021	GAM/SP2
2305	CBPUS013	DFP
2501 2540	CBPUS012	DFP
2740 2740C 2740X 2741	CBPUS024	MVS
2741C 2741P	CBPUS025	MVS
3203 3211	CBPUS012	DFP
3270 3277 3278 3279	CBPUS004	MVS
3284 3286	CBPUS031	MVS
3330	CBPUS001	DFP
3330V	CBPUS015	MSS/XA
3340 3350	CBPUS001	DFP
3350P 3351P	CBPUS003	DFP
3375 3380	CBPUS002	DFP
3420 3420C 3430 3480	CBPUS005	DFP
3505 3525	CBPUS012	DFP
3540	CBPUS032	MICR/OCR
3704 3705 3791L	CBPUS023	MVS
3800	CBPUS011	DFP
3838	CBPUS034	VPSS/XA

Devices	UIM Name	Product
3848	CBPUS041	CUSP
3851	CBPUS015	MSS/XA
3886 3890 3895	CBPUS032	MICR/OCR
4245 4248	CBPUS012	DFP
7770	CBPUS023	MVS
83B3	CBPUS025	MVS
BSC1 BSC2 BSC3	CBPUS026	MVS
CTC	CBPUS014	MVS
DUMMY	CBPUS050	MVS
HFGD	CBPUS035	GAM/SP2
TWX WTTA	CBPUS025	MVS

UIM Restrictions

The following are restrictions on UIM processing.

For each IODEVICE statement, MVSCP polls your UIMs before polling the IBM-supplied UIMs. If you want to provide your own device support for a device that is supported by IBM, you can write a UIM. However, your UIM must do one of the following:

- Assign to the device a generic name different from the generic name specified by the corresponding IBM-written UIM.
- Do not invoke CBPADIT to build a DIT for the device. The DIT built by the IBM-written UIM will be used for the device.

Whenever a UIM sets the IODVUINV flag in the IODV (to indicate that the IODEVICE internal text record is invalid), it must use the CBPZLOG macro to issue an error message. If a UIM service routine sets IODVUINV, the service routine will use CBPZLOG to issue an error message.

A UIM must not call CBPIDFT if IODVUINV is set.

A UIM must conform to the restrictions on the various data areas that it uses.

System Code and MVSCP Data Separation

Associated with each unit control block (UCB) that is built by the MVSCP are the names of the DDT and MLT associated with that device. This association is made by a UIM in the device features parameters (DFP). These names are the two links between the data built by the MVSCP and the device support code that is in a product or a USERMOD.

Since the DDT name and the MLT name form the only links between the device support code and the data built by MVSCP, you can change your device support code and not re-run MVSCP to re-define your I/O configuration(s). You can do this provided:

- The DDT name must not change.
The contents of the DDT may change.
- The MLT name must not change.
The contents of the MLT may change.
- The format and contents of the UCB for the device must not change.

Note: Any of the change in the above bulleted items will result in a change to the UIM for the device.

Using the Sample UIM

A sample UIM is provided in SYS1.SAMPLIB(SAMPUIM). Copy this sample and use it as the basic structure for your UIM. The sample defines a unit record device. Depending on the device you are specifying, the modifications to this sample may be few or significant. SYS1.SAMPLIB(SAMPUIM) is divided into 5 sections:

1. Directions
2. SAMPUIM The Sample UIM itself.
3. UIMJCL The JCL to assemble and link-edit the UIM.
4. SAMPMLT A Sample MLT.
5. MLTJCL The JCL to assemble and link-edit the MLT.

Read and follow the sample carefully when writing your UIM. The sample contains data set names that you must provide. Check for this throughout the sample. This sample supports a unit record device. Devices defined on the IODEVICE statement as UNIT=DUMMY are supported as unit records. Therefore, if you are using the sample to support a DUMMY device, it will require very few changes. Other device support may require a lot of changes, for example, to identify the features. If you are writing a UIM for another type of device or for a more complex unit record device, you may refer to the IBM-supplied UIM for the device that most closely resembles it.

Changing the Sample UIM

The sample UIM is set up for a DUMMY device. When using the SAMPUIM, you may need to change the following fields as indicated:

UNITNM	Put an eight character unit name of the device in the field. This is the value that must be specified on the UNIT parameter of the IODEVICE statement. The unit name may contain trailing blanks.
GNRCNM	Put an eight character generic name of the device in the field. This is the value that can be specified on the UNIT parameter of the JCL DD statements to allocate the device. The unit name may contain trailing blanks.
GNRCPRTI	Put a generic preference value in the field. See Appendix A for the list of IBM provided generic preference values. You may specify any unused preference value to control the order that MVS attempts to satisfy a request for a device from an esoteric device group. This changes the default device preference order that is used with the DEVPREF parameter on the EDT macro. The DEVPREF parameter may be used to override the order that is specified through the default.
ERPINDEX	Set the equate to a decimal ERP index value. The ERP index value was formerly specified on the ERRTAB parameter of the IODEVICE statement. IBM error routines have the values 000 through 219 and 230 through 254. User routines can have values 220 through 229.
GNRCTYPI	Specify the UCB type information. The UCB type information was formerly specified on the DEVTYPE parameter of the IODEVICE statement.
NAMEMLT	Specify the MLT name.

In addition to the above changes, you must change all occurrences of 'SAMPUIM' to the name of your UIM and specify the necessary diagnostic information on the CBPZDIAG macro.

Sample JCL

The JCL for the UIM included in the sample, is to help you assemble and link-edit the UIM. Be sure to include the correct SYSIN and SYSLMOD data set names. The SYSLMOD data set may be SYS1.LINKLIB or you can link-edit it into a data set that you specify as a STEPLIB when you invoke MVSCP.

Sample MLT

The sample for the MLT contains support for a unit record device in which you specify the error recovery procedure. This contains the list of nucleus device dependent module names and the list of link pack area device dependent module names that are required for the unit record device.

To use the sample MLT provided, you must

- Change all occurrences of 'SAMPMLT' to the name you selected for the new MLT.
- Change 'IGE0xxxx' to the name of the ERP for the device.

IBM provides the other modules listed on the NUCL and LPAL parameters. If you specify your own ERP, it must reside in SYS1.LINKLIB or in a data set within the link list concatenation.

The JCL to assemble and link-edit the MLT is also included in SAMPUIIM. Again be sure to change the necessary data set names appropriately.



Chapter 2. Allocation Considerations

Before a job can execute, the operating system must set aside the devices, and space on the devices, for the data that is to be read, merged, sorted, stored, punched, or printed. In MVS, the “setting-aside” process is called **allocation**.

The MVS allocation routines assign **units** (devices), **volumes** (space for data sets), and **data sets** (space for collections of data) according to the *data definition (DD)* and *data control block (DCB)* information included in the JCL for the job step.

When the data definition or DCB information is in the form of SVC 99 text units, the allocation of resources is said to be *dynamic*. Dynamic allocation means you are requesting the system to allocate and/or deallocate resources for a job step while it’s executing. See *MVS/XA SPL: System Macros and Facilities* for details on the use of dynamic allocation.

Serialization of Resources During Allocation

The scheduler component of MVS controls allocation. When the scheduler is setting aside non-sharable devices, volumes and data sets for a job or a step, it must prevent any other job from using those resources during the allocation process. To prevent a resource from changing status while it is being allocated to a job, the scheduler uses **serialization**. Serialization during allocation causes jobs to wait for the resources and can have a major impact on system performance. Therefore, the allocation routines attempt to minimize the amount of time lost to serialization by providing a specific order of allocation processing. See Figure 2-1.

Knowing the order in which the scheduler chooses devices, you can improve system performance by making sure your jobs request resources that require the least possible serialization.

Note: A description of how the collective operation of the allocation routines relates to performance is in *MVS/XA SPL: Initialization and Tuning*, in the discussion of the system resources manager (SRM).

The scheduler processes resource allocation requests in the order shown in Figure 2-1. As you move down the list, the degree of serialization – and processing time – increases.

KINDS OF ALLOCATION REQUESTS	SERIALIZATION REQUIRED
Requests requiring no specific units or volumes; for example, DUMMY, VIO, and subsystem data sets.	No serialization.
Requests for sharable units: DASD that have permanently resident or reserved volumes mounted on them.	No serialization.
Teleprocessing devices.	Serialized on the requested devices.
Pre-mounted volumes, and devices that do not need volumes. Note: Automatic volume recognition (AVR) function reads the volume serial numbers of any volumes that have been pre-mounted on serialized devices.	Serialized on the group(s) of devices eligible to satisfy the request. A single generic device type is serialized at a time.
Online, nonallocated devices that need the operator or MSS to mount volumes.	Serialized on the group(s) of devices eligible to satisfy the request. A single generic device type is serialized at a time.
All other requests: offline devices, nonsharable devices already allocated to other jobs.	Serialized on the group(s) of devices eligible to satisfy the request. A single generic device type is serialized at a time.
Note: Allocation treats MSS devices (3330V) as direct access storage devices.	

Figure 2-1. Processing Order Allocation Requests Requiring Serialization

Improving Allocation Performance

You can contribute to the efficiency of allocation processing throughout your installation, in several ways:

- For **devices**, use the **device preference table**, specified through the MVSCP process, to set up the order for device allocation. See Appendix A for a list of the IBM defined values used in the device preference table.
- For **devices**, use the **eligible device table** to identify your installation's devices as *esoteric groups*, and to group them for selection by allocation processing. See the SCHEDxx member of SYS1.PARMLIB.
- For **volumes**, use the VATLSTxx members of SYS1.PARMLIB to specify volume attributes at IPL.
- For **data sets**, you can prescribe the JCL used for your applications according to the device selection criteria you have set up through the MVSCP process and IPL.

Controlling the Number of DD Statements Allowed

The size of the task input output table (TIOT) determines the maximum number of DD statements allowed per jobstep. The size of the TIOT can range from 16K to 64K. A TIOT that is 16K allows a maximum of 816 DD statements per jobstep. A TIOT that is 64K allows a maximum of 3273 DD statements per jobstep. By changing the size of the TIOT, you can control the maximum number of DD statements allowed in a jobstep. To change the size of the TIOT, specify the hexadecimal value for an integer from 16 to 64 (decimal) in the 8 byte field, DEFTIOTS (offset 23 decimal) in the allocation default CSECT (IEFAB445). See *MVS/XA System Macros and Facilities Volume 1* for more information.

The Device Preference Table

Looking at Figure 2-1, which shows the order in which the scheduler processes allocation requests, you can see that the last three items on the list involve choices among device types. The system uses the **device preference table** to determine the order in which it selects devices to satisfy requests that could apply to more than one generic device type. (“Generic device type” means the general identifier IBM gives a device; for example, the 3330, or the 3800.)

IBM supplies a device preference table that lists the fastest generic device types first. See Appendix A for a list of the IBM defined values used in the device preference table. Through the MVSCP process, you can supply your own list of generic device types on the DEVPREF parameter of the EDT macro. The IBM default device types follow your specifications to form the device preference table.

The order in which the devices are listed in the device preference table is the order in which allocation routines will select them to satisfy your allocation requests. Listing the fastest generic device types at the beginning of the device preference table can result in heavy contention for the fastest eligible devices, because the system tries to allocate the fastest devices first. If you set up your DEVPREF statement to start with generic groups that include many devices (and many channel paths), these devices will be given preference. This will alleviate the contention, and will also give the allocation routines a wide choice within the device groups for the device selection process.

See *MVS/XA MVS Configuration Program Guide and Reference* for details on using the DEVPREF statement, and for the device preference table shipped with the operating system.

To make the best use of device allocation, the installation should decide whether the operator should respond HOLD or NOHOLD to the following message:

IEF433D *jjj-WAIT REQUESTED--REPLY 'HOLD' or 'NOHOLD'*

The system issues this message when the operator requests that the allocation for a specific job wait until the units and/or volumes necessary to complete the allocation are free. The allocation can release the devices that have already been allocated to the job and cannot be shared with other jobs, or can hold the devices (stay allocated) until the job can be completely allocated.

HOLD This means that the system should hold non-sharable devices and volumes already allocated to the job. Select this option if the needed resources are constantly being freed, and allocation requests for other jobs will probably not be held up by the requests made for this job. This job can hold up other requests in two ways: (1) the job has already allocated units needed for another job, or (2) the job's allocation requests are serialized on devices the job is waiting for.

NOHOLD This means that the system should release non-sharable devices and volumes already allocated to the job. Select this option if the needed resources may not be freed for some time, and allocation requests for this job are likely to hold up allocation requests issued for other jobs.

Note: Requests for dynamic allocation are not held up by requests waiting for batch allocation, even though the jobs awaiting batch allocation are holding resources.

The Eligible Device Table

To avoid serializing on specific devices for every allocation request, or serializing on an entire generic group, MVS uses the concept of **esoteric device groups**. Through esoteric device groups, you group together several specific devices under a unique name.

Your allocation requests that use esoteric names (UNIT=TAPE; UNIT=SYSDA) tell the scheduler to choose among all the devices in that esoteric group. If many devices are eligible for an allocation request, the scheduler finds all the eligible devices within the esoteric group, using mount and use attributes and the type of request. If there are still several eligible devices, the scheduler presents those devices to the system resource manager (SRM). The SRM applies its own criteria to the devices indicated by the scheduler, and recommends a device for the data set.

The Use of Esoteric Names

When you establish esoteric group names, users can request different subsets of the generic device types in your installation, thus cutting down on the number of devices serialized by allocation processing. For example, if both batch and time sharing users need 3330s, you could set up two separate esoteric groups with MVSCP, as follows:

```
UNITNAME UNIT=(330,4),NAME=SYSBATCH
UNITNAME UNIT=(334,4),NAME=SYSTSO
```

The effect of these two UNITNAME specifications is that allocations to SYSBATCH serialize only on units 330-333, instead of the entire 3330 generic device group; and, similarly, allocations to SYSTSO serialize only on units 334-337. Figure 2-2 shows the relationships among generic device types, esoteric group names, specific device numbers, and the eligible device groups created by the scheduler's allocation routines.

Note: The esoteric name *SYSALLDA* is a default IBM supplies for the use of the allocation routines. As such, it is a restricted name; do not use it.

Your installation's esoteric device groups are defined in the **eligible device table** (EDT), which is built using the MVS configuration program (MVSCP). You specify the names of the esoteric groups, and the device numbers they include, on the UNITNAME statement for the MVS configuration program.

MVSCP puts the group names and device numbers into the eligible device table, forming allocation groups. In the SCHEDxx member of SYS1.PARMLIB, you identify the eligible device table the system is to use with a particular configuration. Then during IPL, you can specify a particular EDT either by selecting the SCHEDxx member or by allowing the system to use the default.

If you do not specify an EDT ID(xx) in SCHEDxx, the default for the EDT is the IOCONFIG id specified on the SYSCTL frame. If you specify a SCHEDxx member but do not specify an EDT ID(xx) in it, the system defaults to EDT ID(00). See *MVS/XA SPL Initialization and Tuning* for information on the SCHEDxx member and the default.

Users specify the esoteric names on DD statements for input and output data sets. The scheduler uses the allocation groups from the table when selecting devices in response to allocation requests.

The order you use to place your devices in esoteric device groups controls the way the scheduler assigns devices to allocation groups.

Generic Device Types	3800	3420	3350				3330					
Esoteric Group Name	¹	TAPE ²		SYSDA ³				DA1 ⁴		DA2 ⁵		
Device Number	131	151	152	181	182	183	184	190	191	192	193	194
Group⁷ Number	1	2		3		4		5 ⁶	6		7	5 ⁶
Notes:												
¹ The absence of an esoteric group name means that the 3800 can only be requested as UNIT = 3800.												
² TAPE is the esoteric group name for the two 3420 tape drives, with device numbers 151 and 152. Together these form allocation group number 2.												
³ SYSDA is the esoteric group name for two 3350s (device numbers 183 and 184) and five 3330s (device numbers 190-194). These are in allocation group numbers 4, 5, 6, and 7. When UNIT = SYSDA appears on a DD statement, the allocation routines will consider units 183, 184, 190, 191, 192, 193, and 194 as eligible devices.												
⁴ DA1 is the esoteric group name for two 3330 DASDs (device numbers 191 and 192) included in allocation group number 6.												
⁵ DA2 is the esoteric group name for a single 3330 DASD (device number 193). When DA2 is coded on a DD UNIT parameter, this is the only device eligible for that allocation request. It is in group number 7.												
⁶ Device number sequence is not important: allocation group number 5 consists of device numbers 190 and 194, which are included in SYSDA but not in DA1 or DA2.												
⁷ Allocation routines assign allocation group numbers within the esoteric group names you specify in the EDT through the UNITNAME statement for MVSCP. If you do not specify an esoteric group name, then all the devices of the same generic type form a single allocation group.												

Figure 2-2. Relationships among Generic and Esoteric Device Groups

Creating Multiple EDTs

Using the EDT and UNITNAME statements on MVSCP input stream, you can create as many EDTs as your varying device configurations and applications require,

See *MVS/XA: MVS Configuration Program Guide and Reference* for details. On the EDT parameter in SCHEDxx member of SYS1.PARMLIB, you identify the EDT the system is to use. During IPL, you select the SCHEDxx member that contains the EDT needed for a particular configuration.

The Volume Attribute List

In MVS, all direct access and tape volumes have **volume attributes** of **use**, **mount**, and some volumes have a **non-sharable attribute**. Use attributes control how they are allocated. Mount attributes control how or whether they are demounted after being deallocated. Nonsharable attribute controls exclusive use while the volume is demounted and/or mounted. The allocation routines use the volumes' **use** and **mount** attributes in selecting devices to satisfy allocation requests.

Although the system will assign volume attributes in some circumstances, it is primarily your responsibility to decide which attributes the volumes in your installation are to have. The scheduler uses volume attributes in selecting devices and volumes for allocation. Setting use and mount attributes is important to the efficiency of your installation.

During system initialization, you can assign volume attributes to direct access volumes by means of the **volume attribute list (VATLSTxx)**, a member of SYS1.PARMLIB. See *MVS/XA SPL: Initialization and Tuning* for details on including a volume attribute list in IEASYSxx, and on coding the VATLSTxx parmlib member itself.

After an IPL, you can assign volume attributes to both direct access and tape volumes by means of the MOUNT command. The USE= parameter on the MOUNT command defines the use attribute the volume is to have; a mount attribute of *reserved* is automatic.

See *MVS/XA Operations: System Commands* for the details of using the MOUNT command.

Use and Mount Attributes

Every volume is assigned use and mount attributes via a volume attribute list entry at IPL, a MOUNT command, or by the allocation routines in response to a DD statement.

The relationships between use and mount attributes are complex but logical. The kinds of devices available in an installation, the kinds of data sets that will reside on a volume, and the kinds of uses the data sets will be put to, all have a bearing on the attributes assigned to a volume. Generally, the operating system establishes and treats volume attributes as outlined below.

Use attributes

- **Private** — meaning the volume can only be allocated when its volume serial number is explicitly or implicitly specified.
- **Public** — meaning the volume is eligible for allocation to a temporary data set, provided the request is not for a specific volume and PRIVATE has not been specified on the VOLUME parameter of the DD statement.: Both tape and direct access volumes are given the public use attribute.

A public volume may also be allocated when its volume serial number is specified on the request.

- **Storage** — meaning the volume is eligible for allocation to both temporary and non-temporary data sets, when no specific volume is requested and PRIVATE is not specified. Storage volumes usually contain non-temporary data sets, but temporary data sets that cannot be assigned to public volumes are also assigned to storage volumes.

Mount attributes

- **Permanently resident** — meaning the volume cannot be demounted. Only direct access volumes can be permanently resident. The following volumes are always permanently resident:
 - All volumes that cannot be physically demounted, such as drum storage and fixed disk volumes
 - The IPL volume
 - The volume containing the system data sets

In the volume attribute list in SYS1.PARMLIB, you can assign a permanently-resident volume any of the three use attributes. If you do not assign a use attribute to a permanently-resident volume, the default is public.

Note: If 3344s emulating 3340s, and 3350s emulating 3330-1s and 3330-11s, are to be permanently resident, you must include them in the volume attribute list. See *MVS/XA SPL: Initialization and Tuning*.

- **Reserved** — meaning the volume is to remain mounted until the operator issues an UNLOAD command.

Both direct access and tape volumes can be reserved as a result of the MOUNT command; only DASD volumes can be reserved via the volume attribute list.

The reserved attribute is usually assigned to a volume that will be used by many jobs to avoid repeated mounting and demounting.

You can assign a reserved *direct access* volume any of the three use attributes, via the USE parameter of the MOUNT command or the VATLSTxx member of SYS1.PARMLIB, whichever is used to reserve the volume.

A reserved *tape* volume can only be assigned the use attributes of private or public.

- **Removable** — meaning that the volume is neither permanently resident nor reserved. A removable volume can be demounted either after the end of the job in which it is last used, or when the unit it is mounted on is needed for another volume.

You can assign the use attributes of private or public to a removable *direct access* volume, depending on whether or not VOLUME=PRIVATE is coded on the DD statement: if this subparameter is coded, the use attribute is private; if not, it is public.

You can assign the use attributes of private or public to a removable *tape* volume under the following conditions:

– **Private**

- The PRIVATE subparameter is coded on the DD statement;
- The request is for a specific volume; or
- The data set is nontemporary (not a system-generated data set name, and a disposition other than DELETE).

Note: The request must be for a **tape only** data set. If, for example, an esoteric group name includes both tape and direct access devices, a volume allocated to it will be assigned a use attribute of public.

– **Public**

- The PRIVATE subparameter is not coded on the DD statement;
- A nonspecific volume request is being made; or
- The data set is temporary (a system-generated data set name, or a disposition of DELETE).

Figure 2-3 summarizes the mount and use attributes and how they are related to allocation requests.

Volume State	Temporary Data Set	Nontemporary Data Set	How Assigned	How Demounted
	Type of Volume Request			
Public/ Permanently Resident ¹	Nonspecific or Specific	Specific	VATLST entry or by default	Always ² mounted
Private/ Permanently Resident ¹	Specific	Specific	VATLST entry	Always ² mounted
Storage/ Permanently Resident ¹	Nonspecific or Specific	Nonspecific or Specific	VATLST entry	Always ² mounted
Public/ Reserved (tape and direct access)	Nonspecific or Specific	Specific	Direct access: VATLST entry or MOUNT command Tape: MOUNT command	UNLOAD or VARY OFFLINE commands
Private/ Reserved (Tape and direct access)	Specific	Specific	Direct access: VATLST entry or MOUNT command Tape: MOUNT command	UNLOAD or VARY OFFLINE commands
Storage/ Reserved ¹	Nonspecific or Specific	Nonspecific or Specific	VATLST entry or MOUNT command	UNLOAD or VARY OFFLINE commands
Public/ Removable (Tape and direct access)	Nonspecific or Specific	Specific	VOLUME=PRIVATE is not coded on the DD statement. (For tape, nonspecific volume request and a temporary data set also cause this assignment.)	When unit is required by another volume; or by UNLOAD or VARY OFFLINE commands.
Private/ Removable (Tape and direct access)	Specific	Specific	VOLUME=PRIVATE is coded on the DD statement (For tape, a specific volume request or a nontemporary data set also cause this assignment.)	At job termination for direct access; at step termination or dynamic unallocation for tape (unless VOL= RETAIN or a disposition of PASS was specified); or when the unit is required by another volume.

¹Direct access volumes only.
²Note: VARY OFFLINE accomplishes demounting without resetting the permanently-resident flag in the UCB: after a subsequent VARY ONLINE command, the volume will still be permanently resident.

Figure 2-3. Summary of Mount and Use Attribute Combinations

The Nonsharable Attribute

Some allocation requests imply the exclusive use of a direct access device while the volume is demounted and/or mounted. The MVS allocation routines assign the **non-sharable attribute** to volumes that might require demounting during step execution.

When a volume is thus made non-sharable, it cannot be assigned to any other data set until the non-sharable attribute is removed at the end of step execution.

The following types of requests cause the system to automatically assign the non-sharable attribute to a volume:

- A specific volume request that specifies more volumes than devices.
- A nonspecific request for a *private* volume that specifies more volumes than devices. (For MSS, the MSVGP parameter has the same effect as the PRIVATE subparameter.)

- A volume request that includes a request for unit affinity to a preceding DD statement, but does not specify the same volume for the data set. See the discussion of unit affinity in *MVS/XA JCL*.
- A request for deferred mounting of the volume on which a requested data set resides.

Normally, the system will NOT assign the non-sharable attribute to a permanently-resident or reserved volume. The following case is the exception to this rule:

- The allocation request is for more volumes than units, and one of the volumes is reserved. The reserved volume is to share a unit with one or more *removable* volumes, which precede it in the list of volume serial numbers.

For example:

```
DSN=BCA.ABC,VOL=SER=(A,B),UNIT=DISK
```

```
where volume A is removable and
      volume B is reserved
```

In this case, *both* volumes are assigned the non-sharable attribute; neither of them can be used in another job at the same time.

To avoid this situation, do one of the following:

- Specify the same number of volumes as units
- Specify parallel mounting
- Set the mount attribute of volume A as resident or reserved

System Action: Figure 2-4 shows the system action for sharable and non-sharable requests.

The Request is:	The Volume is Allocated:	
	Sharable	Nonsharable
Sharable	allocate the volume	wait ¹
Nonsharable	wait ¹	wait ¹

¹The operator has the option of failing the request. The request will always fail if waiting is not allowed.

Figure 2-4. Sharable and Nonsharable Volume Requests

For more detailed information on how an application's job control language influences the processing of allocation requests, see *MVS/XA JCL Reference*.

For details on how dynamic allocation affects the use attributes of the volumes in your installation, see *SPL: System Macros and Facilities*.

Recovery of Allocated Resources

When an address space abnormally terminates, the allocation routines use the data from the ALLOCAS address space to deallocate all unit control blocks (UCBs) allocated to the failing address space.

If the ALLOCAS address space is not active because of a failure in *that* address space, the allocation routines can deallocate all but sharable direct access UCBs. The units corresponding to these UCBs cannot be varied offline or unloaded until the next IPL, when they are deallocated.

Controlling GRS Requests in MVS/XA

Global resource serialization is an MVS component designed to protect the integrity of local and global resources, particularly data sets on DASD volumes that are shared by two or more systems. GRS allows you to protect the integrity of your installation's resources by serializing the use of them at the data set level.

In order to use GRS to protect global resources, your installation must build a global resource serialization complex that includes the various systems sharing your direct access devices. See *OS/VS2 MVS Planning*:

Global Resource Serialization for more details about GRS and how your installation can use it.

A program can use GRS by issuing a GRS request. These macros (ENQ, DEQ, RESERVE, and GQSCAN) form the interface between a program and the GRS facility. In handling these request, GRS may need to insert or remove an element from the GRS resource queue area.

A system control provides a threshold for the number of requests that GRS will accept from a single address space. This is to prevent one address space (job, started task, or TSO user) from generating enough concurrent requests to exhaust the GRS resource queue area. The threshold value is stored in the GRS vector table (GVT) which resides in the nucleus. For *unauthorized* callers, the threshold value supplied by IBM is 4096 and is stored in the GVTCREQ field of the GVT. For *authorized* callers, the threshold value supplied by IBM is 4111 and is stored in the GVTCREQA field of the GVT. If these values do not suit your installation's needs, you can change them by using the AMASPZAP service aid or SMP. The threshold values apply to each system in your GRS complex so different systems can have different limits.

Note: If you change the GVTCREQ value, then the GVTCREQA should also be changed. It is recommended that the GVTCREQA value be kept at 15 greater than the GVTCREQ value.

GRS keeps track of the number of elements added to or removed from the GRS resource queue area for each address space. This count is recorded in the ASCBCREQ field of the address space control block (ASCB). This count is normally increased according to the ENQ, RESERVE AND GQSCAN macros issued from each address space and normally decreased according to the number of DEQ macros issued. As each ENQ, RESERVE and GQSCAN request is received, GRS determines if increasing the count will exceed the allowed threshold value in the corresponding GVT field. For an unauthorized caller issuing ENQ or RESERVE macros and for all callers issuing the GQSCAN macro, the count in the ASCBCREQ is compared to the value in GVTCREQ. For authorized callers issuing ENQ or RESERVE macros, the count in the ASCBCREQ is compared to the value in GVTCREQA.

In either case, if the threshold will be exceeded, GRS will issue:

- an ABEND 538 for an unconditional ENQ or RESERVE
- a return code of X'18' for a conditional ENQ or RESERVE
- a return code of X'14' for a GQSCAN that normally resulted in a return code of 8. (Return code 8 results when the caller specifies the token parameter in the GQSCAN macro and GRS has additional data to pass back that does not fit into the caller's buffer.)

Note: The GVTCREQA should have a higher threshold value than GVTCREQ. This allows the termination and error routines to issue additional ENQ/RESERVE requests to obtain resources needed for their processing if an unauthorized caller abends with an ABEND 538.



Modifying the System to Fit your Applications

The work of a computer installation is done by its application programs, assisted by the operating system. Setting up the application programs so they will work efficiently with the operating system is one of the major tasks of a system programmer.

This part of *System Modifications* presents discussions of several areas where you can impose installation-wide standards or defaults on your application programs, to enhance their efficiency.

The following books are mentioned in this chapter of *System Modifications*:

- *MVS/XA Debugging Handbook*
- *MVS/XA Diagnostic Techniques*
- *MVS/XA JCL Reference*
- *MVS/XA Operations: JES3 Commands*
- *MVS/XA Operations: System Commands*
- *MVS/XA SPL: Initialization and Tuning*
- *MVS/XA SPL: System Macros and Facilities*
- *MVS/XA SPL: System Management Facilities (SMF)*
- *MVS/XA SPL: User Exits*
- *OS/VS Message Library: VS2 Routing and Descriptor Codes*



Chapter 3. Limiting User Region Size

You may need to enforce a region size limit for application programs. The process of setting limits on user regions involves three elements:

1. The JCL statements for the job, started task, or TSO session;
2. An exit routine that transmits the values specified in the JCL statements;
3. The virtual storage management (VSM) routines that allocate storage in the user region subpools according to the values transmitted.

The system programmer can control this process at two points: when the JCL is coded, and when the exit is taken.

Setting a Default Region Size via JCL

When setting up the guidelines for the JCL to execute your application programs, you need to consider the impact of user region size on the performance of your system.

Based on your judgement, you can supply effective installation defaults that complement those used by virtual storage management.

You can implement your installation defaults by associating selected values with job classes or accounting information, thus controlling the user region limits for varying circumstances.

See “IEALIMIT Processing” for a discussion of the impact of allowing *no* limits on user region size.

For more information about the REGION parameter on the JOB and EXEC statements, see *MVS/XA JCL Reference*.

Setting Default GETMAIN Limits via Exit Routines

The value specified or defaulted on the JCL REGION parameter becomes the size of the user region available to the job, started task, or TSO user.

The value is transmitted, via a user exit, to VSM, which actually allocates the virtual storage in response to GETMAIN macros. Jobs that specify a region value greater than 16 megabytes and that require significant amounts of storage below 16 megabytes, may require an IEFUSI exit to establish limits below 16 megabytes. This is because VSM establishes an unlimited region below 16

megabytes when the region value specified is greater than 16 megabytes. With no limits set below 16 megabytes, a large variable length GETMAIN could use the entire non-extended region.

The IEALIMIT Exit

For MVS/370 and MVS/XA installations, IBM supplies an exit routine module named IEALIMIT that sets region limits for the user private area below 16 megabytes. Virtual storage management routines call IEALIMIT to establish region size and GETMAIN limits for job steps, started tasks, and TSO logons.

You can replace IBM's IEALIMIT with your own routine, if the values set by the IBM version do not fit your installation's requirements.

To replace IEALIMIT, linkedit your own version into the nucleus prior to an initial program load.

Note: You must re-linkedit your routine into the nucleus each time you IPL a different version of the nucleus; all versions of the nucleus contain the IBM-supplied IEALIMIT routine..

To code your own version of IEALIMIT, see *MVS/XA SPL: User Exits*.

The IEFUSI Exit

For MVS/XA installations, the system management facilities (SMF) supplement the function provided by IEALIMIT with a user exit interface, opened during step initiation. At that point, you can supply an exit routine, named IEFUSI, to transmit your desired values to VSM in much the same way that IEALIMIT does.

In IEFUSI, however, you can define region size and GETMAIN limits for the user regions both below and above 16 megabytes. This means you can use the IEFUSI interface to set default limits for applications in both 24-bit and 31-bit addressing modes.

IBM does not supply a routine at the IEFUSI exit; if you do not insert one (or update an existing one), VSM reverts to IEALIMIT (your version or IBM's), and uses IEALIMIT's default values. Because IEALIMIT cannot return values greater than 16 megabytes, VSM attempts to extrapolate meaningful values for the extended private area from those returned for the area below 16 megabytes. See Figure 3-1.

Note: You may already have a routine at IEFUSI to gather information for SMF. If so, you must extend that routine to interface with VSM.

Since VSM uses the information it gets from IEFUSI in the same way it uses that from IEALIMIT, a discussion of how IEALIMIT works — and how VSM uses its values — follows.

IEALIMIT Processing

On entry, IBM's IEALIMIT routine receives the JCL REGION parameter value in register 0 (it is repeated in register 1). IEALIMIT calculates a GETMAIN limit value based on the REGION value, and puts this value in register 1. VSM allocates space in the user's region, in response to GETMAIN requests, according to the values IEALIMIT returns in registers 0 and 1.

The values IEALIMIT returns in registers 0 and 1 are determined by the value specified on the REGION parameter. If the REGION parameter value is 0, or the parameter is absent (and the installation JES default is 0), IBM's IEALIMIT routine receives 0 in register 1. In this case, it in effect sets NO LIMIT on region size or GETMAIN requests, by returning the 0 in register 1.

When VSM receives a value of 0 in register 1 from IEALIMIT, it allows the GETMAIN limit to default to the size of the user private area, and assigns the same value to the region size.

If the REGION parameter value in register 1 is not zero, IBM's IEALIMIT adds 64K to the REGION value and puts the total in register 1. The REGION parameter value in register 0 remains the same.

VSM can then use the value from register 1 as the limit for GETMAINS from user subpools (0-127, 251 and 252), and the value from register 0 as the user region size.

When IEALIMIT provides no limit on the region size, thus forcing VSM to use its default values, so much space within a region might be obtained (via repeated small GETMAINS or a single large GETMAIN) that no space would remain in the private area for the system to use.

This situation is likely to occur when a program issues a variable-length GETMAIN specifying such a large maximum value that most or all of the space remaining in the private area is allocated to the requestor. To avoid an unexpected out-of-space condition, you should require the specification of some region size on JOB or EXEC statements, or make your installation's JES default value nonzero.

The IEFUSI Interface

IEFUSI is the SMF step-initiation user exit interface; it receives control before the initiator starts each job step. In an MVS/370 environment, the IEFUSI exit routine can check job step accounting information, write to a user data set, or create a separate step-initiation record in case of system failure.

See *MVS/XA SPL: System Management Facilities (SMF)* for more information about IEFUSI and other SMF exits.

In MVS/XA, the IEFUSI interface can also include region limit processing for the private area, both below and above 16 megabytes.

You can use the IEFUSI exit to set region size and GETMAIN limit defaults for all your applications. This is possible because VSM uses the IEFUSI values for the private area below 16 megabytes the same way it uses those set by IEALIMIT. The only difference is that IBM's IEALIMIT requests default values by setting registers 0 and 1 to 0, whereas the default indicator for IEFUSI is -1, set by SMF before the exit is taken. The routine at IEFUSI turns on the first bit in the flag word of the VSM parameter list to indicate that VSM is to use its values rather than IEALIMIT's.

Virtual storage management routines use the values developed by IEFUSI or IEALIMIT to determine how to satisfy a user's GETMAIN requests.

Figure 3-1 shows the algorithms that VSM applies to the values set in IEALIMIT and IEFUSI to arrive at user region size and GETMAIN limit values.

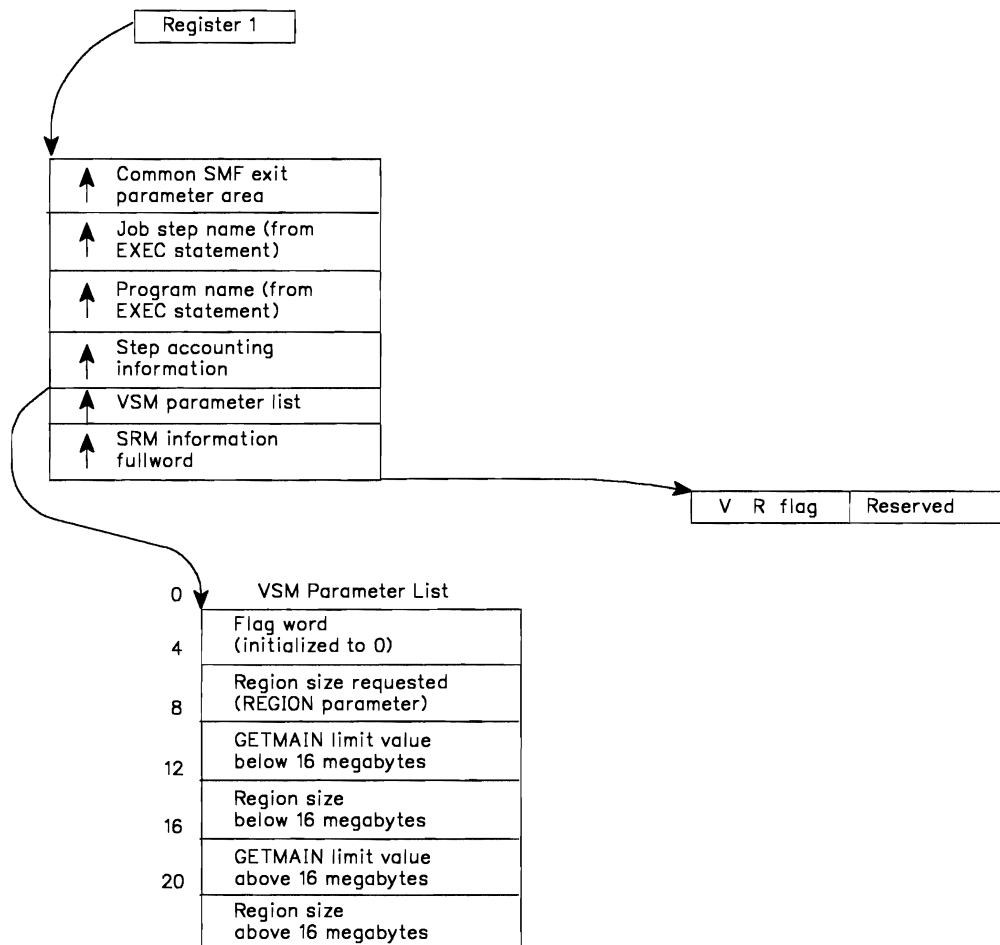
Value From IEALIMIT	Limit Value	Region Size	Extended ² Limit Value	Extended ² Region Size
0 ¹	Size of private area	Limit value	Size of extended private area	Extended limit value
> 0 (Z)	Size of private area or Z, ² whichever is smaller	Limit value or Z, ² whichever is smaller	The smaller of: 1. The greater of 32Mb and REGION request. 2. The size of the extended private area.	Extended limit value
Value From IEFUSI				
	Limit Value	Region Size	Extended Limit Value	Extended Region Size
	<i>If the REGION parameter specifies 0, or defaults to 0.</i>			
-1 ¹	Size of private area	Size of private area	Size of extended private area	Extended limit value
	<i>If the REGION parameter specifies a value greater than 0.</i>			
-1 ¹	The smaller of: 1. Size of private area. 2. REGION request + 64K.	Limit value or REGION request whichever is smaller	The smaller of: 1. The greater of 32Mb and REGION request. 2. The size of the extended private area.	Extended limit value
	<i>Regardless of the REGION parameter specification.</i>			
≥ 0 (X)	Size of private area or X, ² whichever is smaller	Limit value or X, ² whichever is smaller	The smaller of: 1. The greater of 32Mb and X ² 2. The size of the extended private area.	Extended limit value or X, ² whichever is smaller
¹ Requesting that VSM use its defaults. ² Rounded up to a page multiple (4K bytes)				

Note: The algorithm for the extended GETMAIN limit value includes a "max. test" between 32 megabytes (the default extended limit value) and the value specified in IEFUSI. The extended region size will never be less than 32 megabytes.

Figure 3-1. How VSM Arrives at Region Size and Limit Values from Values Set by IEALIMIT and IEFUSI.

Using the IEFUSI interface

To use the IEFUSI VSM interface, you must supply a routine named IEFUSI, and link edit it into LPALIB or an LPALISTxx member of SYS1. PARMLIB. (If your installation already has a routine at exit IEFUSI, you need only update the code to add the VSM interface function.) Your routine puts the values you want VSM to use in setting region size and GETMAIN limits into the fullword fields provided by SMF. Figure 3-2 shows how SMF presents those fields to the IEFUSI exit routine.



Notes:

1. The last four words in the VSM parameter list will be set to 'FFFFFFF' (decimal - 1) on entry to IEFUSI.
2. The SMF parameters can control whether or not the IEFUSI exit is called for various classes of work. If the exit is not called at all, VSM reverts to IEALIMIT.
3. For more details on the calling of the IEFUSI exit, see *MVS/XA SPL: System Management Facilities (SMF)*.

Figure 3-2. Parameters Passed to Exit at IEFUSI by SMF

Your exit routine sets the flag bits in the flag word of the VSM parameter list. The bits in the flag word have the following meanings:

Bit	Value	Meaning
0	0	indicates that IEALIMIT is supplying region limit values.
	1	indicates that IEFUSI is supplying region limit values.
1	0	indicates that VSM should check to see if the requested region fits into the available space below 16 Mb.
	1	indicates that VSM should not check to see if the requested region fits into the available space below 16 Mb.
2	0	indicates that VSM should not check to see if the requested region fits into the available space above 16 Mb.
	1	indicates that VSM should check to see if the requested region fits into the available space above 16 Mb.
3-31		Reserved.

Notes:

- 1. Bit 1 is meaningful to VSM only when the requested region is less than 16 megabytes.*
- 2. Bit 2 is meaningful to VSM only when the requested region is greater than 16 megabytes.*
- 3. Because of compatibility considerations for previous MVS/XA releases, the settings for bits 1 and 2 have opposite meanings. In previous releases, VSM made no checks for free space above 16 megabytes. So, because the flag word is initialized to zero, if the amount of contiguous free space requested is critical for the step to be executed, you must set bit 2 to 1.*

Here is an example of using the IEFUSI interface. You want to limit all jobs in a given step-accounting category to a user region of 4 megabytes below and 4 megabytes above 16 megabytes. You also wish to set a GETMAIN limit of 6 megabytes below 16 megabytes and 48 megabytes above 16 megabytes.

Code the exit routine that will receive control at entry point IEFUSI during step initiation. Your routine would, in this case:

- Examine the step accounting information passed to it to decide whether to apply the IEFUSI limits to this step.
- Set to one the high-order bit in the flag word; this indicates to VSM that IEFUSI, rather than IEALIMIT, is supplying region limit values.
- Examine the current requested region size, to find out if it is already at the desired value.
- Put the GETMAIN limit value for below 16 megabytes (6Mb) in the 3rd fullword of the VSM parameter list.
- Put the GETMAIN limit value for above 16 megabytes (48Mb) in the 5th fullword of the VSM parameter list.
- Put the region size value for below 16 megabytes (4Mb) in the 4th fullword of the VSM parameter list.

- Put the region size value for above 16 megabytes (4 Mb) in the 6th fullword of the VSM parameter list.
- Return to the calling routine at the address in register 14.

VSM applies the following limits when allocating space for the program whose values you set in IEFUSI:

(Assume that the user private area below 16 megabytes is 8 megabytes, and that the extended private area, above 16 megabytes, is approximately 1975 megabytes.)

Limit Value below 16Mb	=	6Mb	(Less than 8Mb)
Limit Value above 16Mb	=	48Mb	(The value from IEFUSI is greater than 32Mb, but less than the extended private area)
Region Size below 16Mb	=	4Mb	(Less than limit value)
Region Size above 16Mb	=	4Mb	(Less than extended limit value)

How VSM Uses the Region Size Value and the Limit Value

The **region size value** determines the amount of storage that can be allocated to a job or step for variable-length GETMAINS, when the minimum amount requested on the GETMAIN is not greater than the storage still unallocated.

The **limit value** is the maximum total storage that can be allocated to a job or step for any combination of GETMAINS. It is, in effect, a second limit on the size of the user's private area, imposed when the region size value has been exceeded.

Figure 3-3 shows how the REGION parameter and the limit value affect both fixed-length and variable-length GETMAINS. The examples that follow illustrate actual allocations based on the interaction of region size and limit values.

Type of GETMAIN	GETMAIN Request in Relation To Region Size and Limit Value	Result
FIXED-LENGTH GETMAIN	Limit value minus currently - allocated space \geq requested amount.	The GETMAIN is satisfied
	Limit value minus currently - allocated space $<$ requested amount.	The GETMAIN fails
VARIABLE-LENGTH GETMAIN	Unallocated space ¹ \geq maximum amount requested.	The maximum amount is allocated.
	Minimum amount requested \leq unallocated space, AND unallocated space \leq maximum amount requested.	All unallocated space in the region is allocated.
	Unallocated space ¹ \leq minimum amount requested	The minimum is allocated unless the limit value would be exceeded, in which case the GETMAIN fails.
¹ Unallocated space is the region size value minus the currently - allocated space. See "Examples of Allocations."		

Figure 3-3. Effect of Region Size and Limit Values on Various GETMAIN Requests

Examples of Allocations Based on Values Set by IEALIMIT or IEFUSI

Assume that application program A has the following characteristics:

Limit value	150K
REGION size value	100K
Space currently allocated	80K

Program A issues the following variable-length GETMAIN requests, in the order indicated (Note that the GETMAIN requests are **cumulative**):

1. Request 5K—10K: 10K is allocated, making currently-allocated space 90K.

Because the amount still unallocated (20K, relative to the region size of 100K), was greater than the maximum amount requested, the maximum amount was allocated.

2. Request 5K—100K: 10K is allocated, making currently-allocated space 100K.

Because the amount still unallocated (10K, relative to the region size) was between the minimum and maximum requested, the unallocated space was allocated.

3. Request 40K—100K: 40K is allocated, making currently-allocated space 140K.

Although the amount still unallocated (0K, relative to the region size) was less than the minimum amount requested (40K), the minimum amount requested would not increase the currently-allocated space beyond the limit value, so the minimum amount was allocated.

4. Request 15K—50K: the GETMAIN fails.

The amount still unallocated (0K, relative to the region size) was less than the minimum amount requested (15K), AND the minimum requested would increase the currently-allocated space to 155K, which exceeded the GETMAIN limit value of 150K.

The region size value is usually set up to be less than the limit value. This will protect against programs that issue variable-length GETMAINS with very large maximums and then do not immediately free part of that space, or free such a small amount that a subsequent GETMAIN (possibly issued by a system service) causes the job to fail.

As an example, suppose that the region size value equals the limit value, and a program issues a variable-length GETMAIN with a maximum of 2 gigabytes - 1. If the GETMAIN is satisfied, all the space in the region up to the limit value will be allocated, and any subsequent GETMAIN that cannot be satisfied from free space in an already-existing subpool will cause the job to fail.

If, however, the region size value is less than the limit value, limit, only space up to the region size value is allocated for the GETMAIN. Thus, an amount of space equal to the limit value minus the region size value remains for subsequent GETMAINS.

Note: For V=R jobs, the REGION parameter is more significant as a limiting value than are the limits set by IEFUSI. You can use the two factors together to control the region size for applications that must run V=R:

- Set the region size value where you want it, via IEFUSI.
- If a REGION parameter specification for a V=R job exceeds the region size value you have set, the job will not be initiated.

Chapter 4. Assigning Special Program Properties to Applications

Program Properties Table

Sometimes, your application programs will need to possess special properties to run as efficiently and securely as possible. For example, an application that requires access to fetch-protected system data will need a system key (0-7) instead of the usual problem program key of 8. Or, for example, an application that cannot run V=R, but must not be swapped out because of real-time considerations, will need to be identified to the system as nonswappable.

In the program properties table (PPT) you can specify the application programs that require special treatment by the system. Each entry in the PPT represents the application program requiring special treatment. To create an entry in the PPT, you use the PPT parameter in the SCHEDxx member of SYS1.PARMLIB. For more information on specifying PPT statements, see *MVS/Extended Architecture System Initialization and Tuning*.

To override an IBM-supplied entry in the PPT, you specify a PPT statement in the SCHEDxx member of SYS1.PARMLIB and use the same program name as the IBM-supplied entry. The system ignores other PPT statements with the same program name and issues message IEF732I.

To add a new PPT entry, you specify a PPT statement in the SCHEDxx member of SYS1.PARMLIB, and specify the desired program name and attributes.

During IPL, you specify the SCHEDxx member the system is to use and the corresponding PPT is then available to the initiators. An initiator scans the PPT to determine which, if any, special properties apply to the program it is initiating.

The CSECT IEFSDPPT includes IEFZB610 and resides in LINKLIB. IEFZB610 maps the PPT header and entry.

Format of the PPT Table Header

A table header precedes the first PPT entry. The table header includes the PPT acronym, version number, length of the header section, length of a PPT entry, the number of PPT entries being used, and the total number of PPT entries.

+0	PPTID		
+4	PPTVERS	res'd	PPTHDRLN
+8	PPTENTLN		PPTUSED
+12	PPTENTS		(Reserved)
+16	PPTMSGAD		
+20	PPTIB650		
+24	RESERVED		
+28			

Format of the PPT Entry

Each entry of the PPT is 16 bytes long and has the following format:

+0	PPTNAME		
+8	PPTBYTE1	PPTKEY	PPTCPUA
+12	PPTPUBYT	PPTORIG	RESERVED

Contents of the PPT Entry

Each PPT entry contains the following fields:

1. PPTNAME
2. PPTBYTE1
3. PPTKEY
4. PPTCPUA
5. PPTPUBYT
6. PPTORIG

The following describes these fields. Following the description of the fields are notes on their usage.

PPTNAME (Program Name) is an 8-byte field for the name specified in the PGM parameter on the EXEC statement for the job or step.

PPTBYTE1 (Program Properties Flags) is a series of bits indicating the special properties to be assigned to the program. The bit settings are:

Bit	Name	Meaning When Set
1... ..	PPTNCNCL	The program cannot be cancelled.
.1.. ..	PPTSKEY	A unique protection key is to be assigned to the program. The key is defined in the next byte of the PPT entry (PPTKEY).
..1.	PPTNSWP	The program is nonswappable.
...1	PPTPRIV	The program is privileged: the address space will not be swapped unless it is in a long wait.
.... 1...	PPTSYSK	The program is a system task, and will not be timed. (The program must be a one-step job started by a START or MOUNT command.)
.... .1..	PPTNSDI	The program does NOT require data set integrity: it will not need exclusive use of any data sets. (The program must be a one-step job.)
.... ..1.	PPTNOPAS	The program can bypass password protection.
.... ...x		Reserved.

PPTKEY (Protection Key) is a 1-byte field whose first four bits indicate the unique protection key to be assigned to the program. A protection key is not assigned unless bit 1 (PPTSKEY) of the preceding field (PPTBYTE1) is on.

PPTCPUA (Processor Affinity Mask) is a 2-byte (halfword) indicating processor affinity. Each bit in the 16-bit mask refers to a corresponding processor identifier (0-F) assigned during system generation. For example, bit 0 corresponds to processor 0. If bit 0 is on, the program is eligible to run on processor 0.

The bit mask should be set to X'FFFF' if affinity is not required. Do NOT set the affinity mask for programs requiring the Vector Facility. For vector facility programs, the control program dynamically manages affinity. If a vector program needs to be in the PPT, for example to set PPTNSWP = 1 for non-swappable, set the processor affinity mask to X'FFFF'.

PPTPUBYT (Preferred Storage Flags) is a one-byte field whose flags indicate whether LSQA and private area fixed pages require frames in preferred storage (nonreconfigurable and non-V = R storage).

Use these flags for programs whose fixed pages could prevent the successful execution of a VARY STOR,OFFLINE command (or could fragment the V=R area) if they were assigned frames in reconfigurable or V=R storage.

The bit settings for PPTPUBYT are:

Bit	Name	Meaning When Set
1..x xxxx	PPT2LPU	Assign all private area short-term fixed pages to preferred frames.
.1.x xxxx	PPT1LPU	Assign all private area long-term fixed pages and LSQA pages to preferred storage frames.
..1x xxxx	PPTN2LP	The system need not assign private area short-term fixed pages to preferred storage frames.
...x xxxx		Reserved.

PPTORIG (PPT Entry Origin) is a one-byte field indicating the origin of a PPT entry. The high-order bit (PPTDEFLT) is set to 1 if the PPT entry is in the IBM-supplied program properties table (IEFSDPPT). PPTDEFLT is set to 0 if the PPT entry originated from a PPT statement in the SCHEDxx member of SYS1.PARMLIB.

Notes on Using the Program Properties Flags (PPTBYTE1)

1. *The special properties represented by the various bit settings in PPTBYTE1 might not be honored by the system. A program is assigned special properties only if it resides in an APF-authorized library and all JOBLIBs and STEPLIBs associated with it are APF-authorized libraries.*

Note: All PPT entries require APF libraries and APF authorization.

2. *The requirements of the initiator have a bearing on whether or not a program needs to maintain data set integrity (bit 5). If one or more data sets requested by a program are not available when the job is to be initiated, the scheduler waits until the job can acquire control of all the data sets it needs. Although the job itself may not require data set integrity, the initiation process for the job does require it.*
3. *Jobs that request the no-data-set-integrity property (bit 5) will not be initiated if BOTH of the following are true:*
 - *The job requests a data set whose name is an alias for a data set that is unavailable during the job's initiation.*
 - *The job contains either a JOBLIB or STEPLIB.*
4. *Jobs requesting the bypass-password-protection property (bit 6) will always receive the property. However, a protected data set cannot be deleted via JCL (that is, by coding a disposition of DELETE) without the password.*

The bypass-password-protection property is turned off when the job enters deallocation processing.

Notes on Using the Preferred Storage Flags

The first two flags (PPT2LPU and PPT1LPU) in PPTPUBYT are meaningful for swappable programs (PPTNSWP=0) that have a special requirement for preferred frames. The third flag (PPTN2LP) is meaningful only for users of the SYSEVENT TRANSWAP. This includes V=R job steps, nonswappable programs, applications using the BTAM OPEN function, and any applications using a system function that issues SYSEVENT TRANSWAP.

The initiator maps the preferred storage flags to corresponding flags in the ASCB. The ASCB flags determine how the system allocates frames to the address space.

1. If PPT1LPU = 1 and PPT2LPU = 0, the initiator sets ASCB1LPU in the ASCB to 1.
2. If PPT2LPU = 1, the initiator sets both ASCB1LPU and ASCB2LPU in the ASCB to 1, regardless of the value of PPT1LPU and PPTN2LP.
3. The value of PPTN2LP is copied to ASCBN2LP.

Notes:

1. *ASCBN2LP merely prevents SYSEVENT TRANSWAP from setting ASCB2LPU to 1 as the address space changes to a nonswappable state. If ASCB2LPU is 1 before the TRANSWAP, it is not reset to 0.*
2. *For a SYSEVENT TRANSWAP, if PPTN2LP=0 then SRM will set ASCB1LPU and ASCB2LPU to 1. This will assign all private area fixed pages and LSQA pages to preferred storage frames. IF PPTN2LP=1, only ASCB1LPU will be set to 1.*

The topic “Examples of Using Preferred Storage Flags” summarizes the effect of the preferred storage flags on the allocation of frames during program execution.

A program need not be nonswappable to have the system assign its fixed pages to preferred storage frames.

Tips on Using the Preferred Storage Flags: TIPS APPLYING TO ALL THREE FLAGS IN PPTPUBYT

For an application program that issues SYSEVENT DONTSWAP, or issues SYSEVENT REQSWAP followed by a SYSEVENT DONTSWAP, do **one** of the following:

- List the program in the PPT with the first two preferred storage flags set on (PPT1LPU = 1, PPT2LPU = 1).

This allows the program to be attached as swappable, but all LSQA and private area fixed pages will be assigned preferred frames during the entire job step.

- Remove SYSEVENTs REQSWAP and DONTSWAP from the program. List the program in the PPT as nonswappable (PPTNSWP = 1) and set PPTN2LP = 0.

This allows the program to be attached as nonswappable, and all LSQA and private area fixed pages will be assigned preferred frames during the entire job step.

An I/O device requiring operator intervention can interfere with taking storage offline by fixing pages in reconfigurable storage. An example of this is a printer requiring action to be taken or a tape unit with a mount pending. Until the required action is completed, the storage associated with the I/O operation cannot be taken offline. This problem *cannot* be bypassed through the use of preferred storage flags.

The system ignores all three flags if any non-APF-authorized JOBLIBs or STEPLIBs are defined in the JCL for the job step. All PPT entries require APF libraries and APF authorization.

TIPS APPLYING TO FLAGS PPT1LPU AND PPT2LPU

PPT1LPU and PPT2LPU are intended for use with authorized swappable programs that issue SYSEVENT DONTSWAP to become nonswappable for relatively short periods (rather than setting PPTNSWP = 1).

Use of the preferred storage flags forces the program's private area fixed pages and LSQA pages into preferred storage frames, thus ensuring that they will not prevent taking storage offline.

TIPS APPLYING TO FLAG PPTN2LP

PPTN2LP has meaning only for programs for which SYSEVENT TRANSWAP is issued. TRANSWAP causes the transition of the address space to a nonswappable state. TRANSWAP performs the same function as SYSEVENT DONTSWAP and also ensures that preferred storage is used whenever necessary.

1. The initiator issues TRANSWAP for V = R job steps and nonswappable programs (PPTNSWP = 1).
2. The BTAM OPEN routine issues TRANSWAP.

PPTN2LP should be set to 1 when a program's short-term fixed pages do not need to be assigned to preferred storage frames. That is, the program's short-term fixes are indeed short-term fixes and can be allowed in reconfigurable storage.

The PPT1LPU and PPT2LPU bits should both be set to 1 when the preferred storage requirements for a nonswappable user are unknown. This will ensure that all fix requests and LSQA requests will get preferred storage.

Examples of Using Preferred Storage Flags

- The following example shows the effect of setting preferred storage flags for the nonswappable program JES2.: The JES2 entry in the PPT would include the following bit values:

```
PPTNSWP = 1
PPT1LPU = 0
PPT2LPU = 0
PPTN2LP = 1
```

These values indicate that the program is nonswappable and that the short-term fixes can be allowed in reconfigurable storage. After the initiator issues a TRANSWAP and attaches JES2, the ASCB flags are set as shown below:

```
ASCB1LPU = 1
ASCB2LPU = 0
ASCBN2LP = 1
```

These values result in LSQA and long-term fixed pages in preferred storage only. Short-term fixed pages are allowed in reconfigurable storage.

- The following example shows the effect of setting the flags for a swappable program that issues SYSEVENT DONTSWAP.

The program's entry in the PPT would include the following bit values:

```
PPTNSWP = 0
PPT1LPU = 1
PPT2LPU = 1
PPTN2LP = 0
```

These values indicate that the program is swappable and that all fixed pages and LSQA pages must be in preferred storage. The initiator attaches the program as swappable; the ASCB flags are set as follows:

```
ASCB1LPU = 1
ASCB2LPU = 1
ASCBN2LP = 0
```

The program can then issue DONTSWAP, being assured that its fixed and LSQA pages are in preferred storage and will not prevent storage from being taken offline.

Common Usage of the Preferred Storage Flags

Following is a summary of the most common uses of the PPT preferred storage flags:

PPTNSWP	PPT1LPU	PPT2LPU	PPTN2LP	Effect on Program
1	-	-	0	The initiator makes the address space nonswappable via the SYSEVENT TRANSWAP prior to attaching the job step. LSQA and all private area fixed pages are in preferred storage.
1	-	-	1	Same as preceding case except short-term fixed pages are allowed in reconfigurable or V = R storage.
0	1	1	-	The initiator attaches the job step as swappable. LSQA and all private area fixed pages are in preferred storage. In this case, the program can issue DONTSWAP and be assured that its fixed pages will not prevent reconfiguring storage.

Note: A dash (-) indicates that the setting of the bits is irrelevant.

Updating the PPT

You use PPT statements in SCHEDxx member of SYS1.PARMLIB to update the PPT. See *MVS/Extended Architecture Initialization and Tuning* for the syntax of the SCHEDxx member.

Note: A TCAM Message Control Program (MCP) will not operate unless its name is in the PPT. TCAM OPEN routines must run in key 6; they will abnormally terminate any caller that is not initiated in key 6.



Chapter 5. Creating Your Own Resource Managers

When the applications in your installation include programs that allocate resources for their own use or for the use of programs they control, they may have to include routines that “clean up” the queues and control blocks associated with the resources, before returning to their calling routines.

MVS provides system resource managers to clean up during termination of its own tasks and address spaces; you can provide similar routines to do the same for your application tasks and address spaces.

The responsibilities of a resource manager are:

- **At task termination:** remove all traces of the fact that the TCB for the terminating task was connected to, allocated to, or associated with, the resources it used. Each resource (data set, volume, device) is left in such a state that another task in the address space or system can reuse it.
- **At address space termination:** release all system queue area and common service area control blocks obtained for the use of the terminating address space. All buffers, bit settings, pointers, and so on relating to the address space are reset to make the system appear as if the ASID and ASCB for the terminating address space never existed.
- **At entry:** establish a recovery environment (ESTAE or ESTAI, or ETXR) to protect itself against errors during its own processing. If the recovery routine is an ESTAE type, the ESTAE macro must include the TERM = YES option.

Installation-Written Resource Managers

Installation-written resource managers can perform the same type of work as a system-provided resource manager; they can also include any special processing your installation requires.

The recovery termination manager (RTM) invokes an installation-written resource manager whenever a task or an address space terminates, either normally or abnormally. The module names of all the system-provided resource managers are known to RTM; RTM invokes them after all installation-written resource managers have completed processing.

When RTM invokes an installation-written resource manager at task termination, the resource manager executes under an RB in the terminating TCB's address space.

When RTM invokes an installation-written resource manager at address space termination, it executes in task mode in the master scheduler's address space. In either case, the resource manager gets control in key 0, supervisor state, with no locks held.

The Resource Manager Parameter List

The interface between an installation-written resource manager and recovery termination management is the resource manager parameter list (RMPL), which RTM supplies to communicate with the resource manager.

The RMPL tells the resource manager why it was invoked and provides information for its use during processing. RMPL fields indicate, for example, whether the resource manager is being invoked during task termination or address space termination, and whether the termination is normal or abnormal.

To access the contents of the RMPL, the resource manager routine must include the IHARMPL mapping macro instruction, which provides the field names and describes their content and use. Detailed information on the name, offset, and meaning of each field in the RMPL appears in the *MVS/XA Debugging Handbook*.

Figure 5-1 lists the names and meanings of some of the key fields in the parameter list.

On entry to the resource manager, register contents are:

Register	Contents
1	Pointer to a 4-byte field containing the address of the resource manager parameter list (defined in your routine by the IHARMPL mapping macro instruction).
13	Pointer to a standard save area (72 bytes).
14	Return address.
15	Entry point address in the resource manager.
0,2-12	Unpredictable

Your resource manager must save and restore registers 0-14; use register 15 to pass a return code back to RTM. The possible return codes are:

0	Indicates successful processing
4	Indicates unsuccessful processing

RMPLTYPE	If set to 1, indicates that the resource manager is being invoked during abnormal termination; if set to 0, indicates that the resource manager is being invoked during normal termination.
RMPLTERM	If set to 1, indicates that the resource manager is being invoked during address space termination; if set to 0, indicates that the resource manager is being invoked during task termination.
RMPLASID	Indicates the ASID associated with the terminating task or address space.
RMPLASCB	Indicates the address of the ASCB associated with the terminating task or address space.
RMPLTCBA	Indicates the address of the terminating TCB (for task termination) or contains zeros (for address space termination).
RMPLRMWA	Indicates the address of a 64-byte work area for use by your resource manager.

Figure 5-1. Some Key Fields in the Resource Manager Parameter List (RMPL)

Adding an Installation-Written Resource Manager

To add your own resource manager routines for installation applications, place their names in the CSECT IEAVTRML, which is provided by MVŠ.

Initially, IEAVTRML consists of four 12-byte entries, each containing zeros. You can modify each of the first three 12-byte entries (using the AMASPZAP service aid) to contain a module name in the first eight bytes; the last four bytes of each entry are reserved and always contain zeros. The last entry must also contain all zeros, to indicate the end of the list. A typical entry for the CSECT might be:

```
DC    CL8 'MODULENM'  
DC    XL4 '00'
```

To add the names of more than three installation-written resource managers, create an entry for each module and a final entry that contains all zeros. Then assemble your modified IEAVTRML and use the modified CSECT to replace the existing IEAVTRML module in load module IGC0001C in SYS1.LPALIB. Place each installation-written resource manager routine in SYS1.LINKLIB (or a library concatenated to SYS1.LINKLIB via a LNKLSTxx member of SYS1.PARMLIB) or SYS1.LPALIB. If every routine named in IEAVTRML is not present in one of these libraries, the IPL will fail.

Chapter 6. Executing DAT-off Code in MVS/XA

In the MVS/XA environment, code that runs with DAT (dynamic address translation) off must reside in the DAT-off nucleus. You invoke the DAT-off code using the DATOFF macro, which controls the dynamic address translation facility.

To add DAT-off code to the DAT-off nucleus, and execute the code, follow these steps:

1. Create a separate module containing the code that runs with DAT off, as follows:
 - Use entry point IEAVEUR_n, where *n* is a number from 1 to 4. MVS/XA reserves four entry points in the DAT-off nucleus for user code.
 - Give the module AMODE 31 and RMODE ANY attributes.
 - Make sure the DAT-off code does not alter register 0; it contains the return address to the routine that issues the DATOFF macro.
 - Use BSM 0,14 as the return instruction.
2. Linkedit your DAT-off module (IEAVEUR_n) into SYS1.NUCLEUS, the DAT-off nucleus data set. The member name is IEAVEDAT; input to the linkage editor must include an ENTRY control statement for entry point IEAVEDAT.
3. Within a DAT-on routine, code a DATOFF macro to invoke the module created in step 1:

```
DATOFF INDEX=INDUSRn
```

The suffix of the index (*n*) is the same as the suffix of the DAT-off module's entry point, IEAVEUR_n. See *MVS/XA SPL: System Macros and Facilities* for details on coding the DATOFF macro.

The DATOFF macro branches to a routine in the PSA that turns DAT off and branches to the DAT-off routine (IEAVEUR_n) in the DAT-off nucleus. Return from IEAVEUR_n is likewise through the PSA routine, which turns DAT on and returns to the DAT-on code.

Figure 6-1 shows how the DATOFF macro instruction works with your DAT-off code.

Note: You will need to re-linkedit your IEAVEURn module(s) into the DAT-off nucleus if you re-sysgen the MVS base control program.

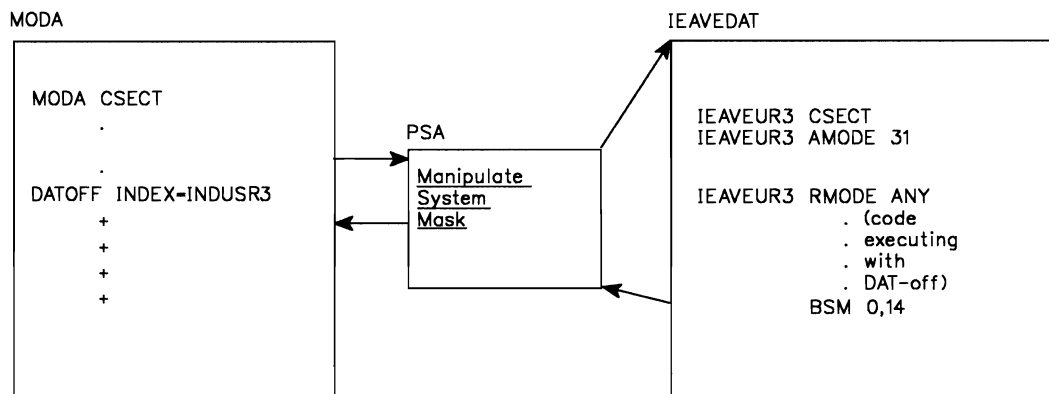


Figure 6-1. Using the DATOFF Macro to Execute DAT-off Code

Chapter 7. Controlling System Messages and the System Log

The operating system communicates with your installation through messages written to the various consoles, user terminals, and printers. The installation communicates with the operating system through responses to its messages and through commands entered on the operators' consoles.

You can control the communications between the system and your installation by controlling the routing of system messages, by suppressing unnecessary messages and by changing the text of certain messages to provide additional information.

Two subcomponents of MVS can help you control system messages. They are:

- *Multiple-console support* (MCS), which routes system messages throughout the installation
- The *message processing facility* (MPF), which provides a means to identify messages you want
 - Suppressed
 - Retained or not retained through the action message retention facility (AMRF)
 - Passed to a user-specified WTO exit for additional processing.

You can also control the routing of system messages to the MVS operators' consoles with the LEVEL keyword of the CONTROL V command. Use the LEVEL keyword to specify the importance levels of messages that a particular console can accept. For instance, you can specify that a certain console receive some combination of the following:

- Immediate action messages
- Eventual action messages
- Critical eventual messages
- Broadcast messages
- Informational messages
- Messages requiring a reply (WTOs)

MVS/XA Operations: Systems Commands describes in detail the syntax of the LEVEL keyword of the CONTROL V command.

In addition to controlling the messages issued by the system, you can control the *system log* data sets, where messages and other information are written via the WTL (write to log) macro and the LOG operator command. See "Controlling the System Log."

Controlling System Messages

The system uses the WTO (write-to-operator) and WTOR (write-to-operator-with-reply) macros to issue messages to the operators' consoles. Application programs can also issue WTO and WTOR macros.

Multiple-console support (MCS) routes messages to different functional areas of an installation, according to the type of information the message contains. For MCS, a "functional area" is defined as one or more consoles that are doing the same kind of work. Some examples of functional areas are:

- The tape library
- The direct access pool
- The system programmer

Each WTO and WTOR macro includes one or more routing codes, used to determine the destination of the message, and one or more descriptor codes, which indicate why the message is being issued. The 128 routing codes available for WTO/WTOR messages are:

Code	Meaning
1	Master console action
2	Master console information
3	Tape pool
4	Direct access pool
5	Tape library
6	Disk library
7	Unit record pool
8	Teleprocessing control
9	System security
10	System error/maintenance
11	Programmer information
12	Emulators
13-20	Available for customer use
21-28	Reserved for subsystem use
29-41	Reserved for IBM use
42	General information about JES2 or JES3
43-64	Reserved for JES use
65-96	Message is associated with a particular processor
97-128	Message is associated with a particular device

When a message is ready for writing, the routing codes assigned to the message are compared with the routing codes assigned to the consoles. The message is sent to each console whose code matches one of those assigned to the message.

The WTO/WTOR macro also includes descriptor codes, which determine how the message is displayed; the color, highlighting, or intensity of the message on the screen; which part of the screen is used for the message; whether or not the message may be rolled off the display. The descriptor codes for a given message are selected according to the condition that prompts the message, and the kind of action or response required of the operator. The descriptor codes for WTO/WTOR messages are:

Code	Meaning
1	System failure
2	Immediate action required
3	Eventual action required
4	System status
5	Immediate command response
6	Job status
7	Application program/processor
8	Out-of-line message
9	DISPLAY, DEVSERV, MONITOR command response
10	TRACK command response
11	Critical eventual action required
12-16	Reserved

See *MVS/XA Message Library: Routing and Descriptor Codes* for more detailed discussion of the codes listed here.

Using a Message-Routing/Processing Exit Routine

When the standard system messages and routing codes are not appropriate for the console configuration in your installation, you can modify them with your own exit routines at the WTO/WTOR user exit points: IEECVXIT, IEAVMXIT, and one or more user-specified WTO exits. Your exit routines receive control before the routing of each message. A routine at IEECVXIT can modify only the message's routing or descriptor codes; however, your routines at IEAVMXIT and the user-specified exits can modify the text of the message and do other processing as well as doing rerouting where necessary. The system then routes the modified message according to the new or modified routing codes.

IEECVXIT Routines

The message-routing routine you supply at IEECVXIT uses the multiple console support function to control which console receives a given message.

Because the text of a message might subsequently be modified by the job entry subsystem, or another subsystem, your exit routine at IEECVXIT can examine, but not modify, the text of a message. Also, this exit does not receive control for all messages. When the WTO/WTOR macro that creates a message includes both the MSGTYP parameter and the JOBNAMES, STATUS or Y parameter, the system does not take the IEECVXIT exit. Similarly, for multiple-line WTOs (including status displays) and messages that are being returned to a requesting console (a response to a DISPLAY A command, for example), the system bypasses the IEECVXIT exit. Because this exit cannot modify message texts and does not receive control for all messages, your use of it is limited. You must use

the general-purpose IEAVMXIT and the user-specified WTO exits for processing the other messages and for additional processing on the messages that go to IEECVXIT.

IEAVMXIT and User-Specified Exit Routines

The general-purpose IEAVMXIT and the user-specified WTO exits allow your routines not only to examine the text and modify the routing and descriptor codes of a message, but also to modify the text and do other processing. Either your general-purpose IEAVMXIT exit routine or your user-specified WTO exit routine receives control for all messages after the exit routine at IEECVXIT. The general-purpose routine does processing common to a large number of messages; the user-specified exit routine does message-specific processing. The exits receive control during the MPF processing.

The IEAVMXIT exit and user-specified WTO exits are mutually exclusive. That is, if, for a particular message ID, you have not named a user-specified WTO exit routine to do specific processing, IEAVMXIT, the general-purpose WTO user exit, receives control.

Because these routines receive control after the IEECVXIT exit routine has completed processing, you need not rewrite your existing IEECVXIT exit routines. However, it is strongly recommended that you replace them with either an IEAVMXIT or a user-specified WTO exit routine that can not only perform the same functions but do additional processing (if desired) on a message.

These routines, like the IEECVXIT exit routine, must be reentrant and serially reusable. They should not:

- Use macros whose expansions store information into an online parameter list,
- Enter an MVS WAIT
- Invoke any service that can issue an MVS WAIT.

User WTO/WTOR Processing

You can set up user exit routines to do the following:

- Reduce message traffic by *selectively* suppressing (filtering) occurrences of messages. (MPF suppresses *all* occurrences of a particular message.)
- Perform error thresholding to reduce operator load.
- Reduce message traffic at specific consoles by redirecting a portion of the traffic.
- Handle the common requests (WTORs) from the system, thereby reducing the need for operator interaction.
- Selectively collect messages at a specific console for a particular purpose.
- Change the presentation of a message to make it more understandable by the operations staff.

With your general-purpose WTO user exit routine, or any message-specific WTO user exit routine(s), you can:

- Override MPF suppression
- Change the message text, routing and descriptor code(s), and the console ID on which the message is to be displayed
- Queue the message to a particular active console, to a particular console unconditionally (even if the console is not active), or queue it by routing codes only
- Force the message to go to the hardcopy log, not to go to the hardcopy log, or to go only to the hardcopy log
- Broadcast a message to all consoles or not broadcast a message already being broadcast
- Issue supervisor calls (SVCs) such as MGCR and WTO
- Delete the message (Except for a WTOR, prevent it from appearing anywhere - hardcopy or display console)

Note: If you try to delete a WTOR, it will be suppressed instead. Then you can view it only by using the DISPLAY R command.

- Indicate whether AMRF should retain or not retain an action message
- Reply to a WTOR
- Suppress a WTOR

See *MVS/XA User Exits* for a more complete description of user-exit capabilities and coding interface.

Inserting A WTO/WTOR Exit Routine into the Control Program

As part of the basic control program, IBM supplies a dummy module at exit point IEECVXIT that does not modify the routing codes assigned to WTO/WTOR messages. If you only want to modify the routing codes, you can code your own IEECVXIT exit or, preferably, a general-purpose IEAMVXIT or a user-specified WTO exit routine. However, If you want to change the message text, or do other types of processing on the messages, you **must** code your own general-purpose IEAVMXIT exit and one or more message-specific WTO/WTOR exit routines.

The procedure for inserting your exit routines into the base control program differs depending on the type of WTO exit involved.

IEECVXIT Routine

Before system generation, you can insert your own message-routing routine into the basic control program by replacing the dummy module with your own. Use the linkage editor to include the module (named IEECVXIT) in the library named SYS1.AOSC5.

After system generation, the dummy module is CSECT IEECVXIT, in load module IEECVXIT, residing in SYS1.LPALIB. Use the linkage editor to replace the CSECT with your own routine.

See *MVS/XA SPL: User Exits* for the restrictions and details of coding your message-routing exit routine for IEECVXIT.

IEAVMXIT

You can insert your IEAVMXIT exit routine in the control program by:

- Link-editing it into SYS1.LINKLIB. You must use 31-bit addresses in the routines and assemble them with AMODE 31; the RMODE must be ANY.
- Activating it with the CONTROL command: K M,UEXIT=Y

Note: If you have already link-edited a general-purpose routine into SYS1.LINKLIB before this IPL, you need not perform the above steps. The system will activate the general-purpose exit.

User-specified WTO Exit Routines

You can insert user-specified WTO exit routines into the control program by:

- Link-editing them into SYS1.LINKLIB. You must use 31-bit addresses in the routines and assemble them with AMODE 31; the RMODE must be ANY.
- Putting the name of each message-specific WTO exit routine you write into a MPFLSTxx parmlib member. You must put the name of the WTO exit routine in the USEREXIT parameter of the message ID entry for each message the routine is to process.
- Activating the MPFLSTxx member with SET MPF=xx

Note: To activate the MFPLSTxx during IPL, you must have link-edited the exit routines into SYS1.LINKLIB, and you must have specified the MPF(xx) keyword on the INIT statement in the CONSOLxx member.

Replacing a WTO/WTOR Exit Routine Without a re-IPL

There may be times when you need to replace a general-purpose or message-specific exit routine either because of adding function(s) to the routine or because a routine abended when processing a particular message. Depending on whether the routine is a general-purpose or message-specific routine, the procedures are as follows:

IEAVMXIT: If you want to replace a general-purpose routine with a fresh copy, you must do the following:

- Link-edit the new copy of the exit routine into SYS1.LINKLIB.
- Refresh LLA with the MODIFY LLA,REFRESH command.
- Reactivate the exit routine using the K M,UEXIT=Y command.

User-Specified exit: If you want to replace a user-specified exit routine with a fresh copy, you must do the following:

- Link-edit the new copy of the message-specific routine into SYS1.LINKLIB
- Refresh LLA with the MODIFY LLA,REFRESH command
- Reactivate the exit routine using the SET MPF=xx command

See *MVS/XA SPL: User Exits* for the restrictions and details of coding your routines for IEAVMXIT and user-specified exits.

See *MVS/XA SPL: Initialization and Tuning* for details and syntax rules that apply to the creation of MPFLSTxx members of SYS1.PARMLIB.

The Hardcopy Log

The hardcopy log contains messages and responses to commands. The log is often written to an output-only device such as a printer, but it can be buffered on the system log data set (SYSLOG). Often, the system log (see “Controlling the System Log”) and the hardcopy log are the same; in many installations, the terms are used interchangeably.

Note: In a JES3 installation, both logs are treated differently than they are under JES2. See *MVS/XA JES3 Commands* for more information.

The number and type of messages recorded on the hardcopy log are optional. Unless you specify otherwise, the hardcopy log includes those messages with routing codes 1, 2, 3, 4, 7, 8, 10 and 42. You can redefine the routing codes for the hardcopy log through the CONSOLxx member of SYS1.PARMLIB and after IPL, by means of the VARY operator command. See *MVS/XA SPL: Initialization and Tuning* for information.

If you modify message routing codes in an IEECVXIT, IEAVMXIT, or a user-specified exit routine, the system normally records the messages with modified routing codes in the hardcopy log, provided the log is active. However, the installation’s IEAVMXIT and user-specified WTO user exit routine(s) can totally delete some messages. The hardcopy log is activated via the VARY command, or by the operating system when MCS and/or JES3 is active. See *MVS/XA System Commands* for more information.

You can direct selected groups or types of messages or commands to the hardcopy log, using your WTO/WTOR exit routine. If commands are recorded on the hardcopy log, the system also records command responses.

Suppressing the Display of Selected Messages

To cut down on the number of messages the operator must deal with, you can suppress the display of certain messages. The message processing facility, an extension of the multiple-console support function, builds a table in the common service area containing IDs of messages to be suppressed.

You can suppress any messages except the following types:

- Action messages (descriptor codes 1, 2, 3, and 11)
- WTORs
- Command responses (descriptor code 5 and MCSFLAG = RESP)
- JES3 action messages

Both the message processing facility and the hardcopy log must be active for the system to allow suppression of messages. This is because suppressed messages are written to the hardcopy log, and are flagged with an indicator (For JES3 only, & is the IBM default; you can choose any indicator.).

To suppress the display of messages, you give MPF a list of message IDs for the table it builds in the CSA.

Create MPFLSTxx members of SYS1.PARMLIB, each containing a list of IDs and/or prefixes for messages you want suppressed. The messages to be suppressed might be selected according to different configurations of your system, or according to the job mix.

The operator activates MPF using the SET command (MPF = xx). He can display the table of message IDs using the DISPLAY command.

You can also use a general-purpose IEAVMXIT exit and user-specified WTO exit routines, which get control during MPF processing, to suppress messages that MPF can not suppress, or to override MPF suppression. The hardcopy log does **not** have to be active for you to do this. See *MVS/XA SPL: User Exits* and *MVS/XA SPL: Initialization and Tuning* for details on the syntax and use of these WTO exits.

Controlling the System Log

The system log is an integral part of MVS. It consists of dynamically-created data sets that reside in the primary job entry subsystem's spool space and record the communications among problem programs, operators, and the operating system.

The system log often includes the hardcopy log (see "The Hardcopy Log"). When it does, the contents of the hardcopy log are stored on the spool data set rather than being printed by a unit record device.

The system log contains operating data entered by means of the write-to-log (WTL) macro, issued by both system functions and problem programs. The log includes the following information:

- Job time, job step time, and data from the JOB and EXEC statements of a job that has ended
- Descriptions of unusual events recorded by the operator via the LOG command
- Write-to-operator (WTO) and write-to-operator-with-reply (WTOR) messages
- Accepted replies to WTOR messages
- Commands issued through operators' consoles and the input stream, and commands issued by the system

Note: The exact format of the output from the WTL macro varies depending on whether the job entry subsystem is JES2 or JES3. See *MVS/XA JES3 Commands* for information about the format of the WTL message in a JES3 environment.

If you do not modify system log operation, the operating system automatically allocates the system log data set during IPL as a class A SYSOUT data set (SYSLOG).

Modifying the System Log

Once activated, the system log keeps track of the number of entries it receives by counting the WTL macros executed against it. After a certain number of WTLs, the system closes and dynamically deallocates the full log data set and allocates and opens a new system log data set.

You can alter the default operation of the system log to control the processing of the log data sets. You can change the SYSOUT class of the log data sets and the number of WTLs received before switching log data sets.

The processing of the log data sets can be controlled from the operator console or via the SYS1.PARMLIB member IEASYSxx. See *MVS/XA SPL: Initialization and Tuning* for information on modifying the system using members of SYS1.PARMLIB.

Your IEASYSxx parmlib member is included in the system during IPL, in response to the request to specify the system parameters. The following system parameters initialize or alter the system log control values:

- LOGLMT — controls the number of WTLs received before the system switches data sets
- LOGCLS — controls the SYSOUT class of the system log data set

The LOGLMT value must be a six-digit number in the range 000001-999999. An all-zero LOGLMT value results in the system default of 500. When choosing the LOGLMT value, consider the following:

- Whether the system log is defined as MCS hardcopy, and
- Whether the system log data is sufficiently critical to the system to justify frequent allocating, switching, and queuing to a SYSOUT class.

The LOGCLS value must be one alphanumeric character. The default value is class A.

The following example shows the correct format for including the LOGLMT and LOGCLS parameters in an IEASYSxx member of SYS1.PARMLIB when specifying the system parameters during IPL:

```
LOGLMT=004852,LOGCLS=L
```

These two parameters would cause the system log task to switch data sets after 4852 WTLs, and the job entry subsystem to queue the current data set for SYSOUT processing with class L.

From the console, the operator can control the processing of the system log with the WRITELOG system command. For example, the operator can issue a WRITELOG command with the START operand after a system failure or after a WRITELOG command with the CLOSE operand.

In addition, the operator can use the WRITELOG command to force the system to queue the system log data set for printing before the LOGLMT threshold is reached. For further information about the system log and the WRITELOG command, see *MVS/XA System Commands*.

Chapter 8. Updating the Master Job Control Language Data Set

You may need to change the master scheduler's JCL. If, for example, your terminal interface is not TSO, you will need to modify the master JCL data set.

The master job control language data set (CSECT name MSTJCL00 and load module name IEEMSJCL) is a non-executable module. The master scheduler loads this module into the system during the initialization of the master scheduler. The IBM-supplied IEEMSJCL module contains data definitions (DD) statements for all system input and output data sets that are necessary for communication between the operating system and the primary job entry subsystem. The primary job entry subsystem can be JES2 or JES3.

IEEMSJCL does not contain the START command that starts the primary job entry subsystem during master scheduler initialization. You define the primary job entry subsystem on the PRIMARY parameter in the IEFSSNxx member of SYS1.PARMLIB.

You can modify the master JCL data set to include START commands for other subsystems, along with DD statements necessary to communicate with them. You can also delete DD statements that do not apply to your installation's interactive configuration.

IEEMSJCL Example

Figure 8-1 shows IEEMSJCL as it exists before it is loaded into the system.

```
DC CL80'//MSTJCL00 JOB MSGLEVEL=(0,0)'  
DC CL80'// EXEC PGM=IEEMB860,DPRTY=(15,15)'  
DC CL80'//STCINRDR DD SYSOUT=(A,INTRDR)'  
DC CL80'//TSOINRDR DD SYSOUT=(A,INTRDR)'  
DC CL80'//IEFPDSI DD DSN=SYS1.PROCLIB,DISP=SHR'  
DC CL80'//IEFPARM DD DSN=SYS1.PARMLIB,DISP=SHR'  
DC CL80'//SYSUADS DD DSN=SYS1.UADS,DISP=SHR'  
DC CL80'//SYSLBC DD DSN=SYS1.BROADCAST,DISP=SHR'  
DC CL80'/*'
```

Figure 8-1. IEEMSJCL Data Set

Included in IEEMSJCL are the DD statements needed to define the internal reader data sets for started task control and TSO logons. Also defined are the system data sets (SYS1.UADS and SYS1.BROADCAST) used in TSO logons and terminal communications.

IBM-supplied Sample of Master JCL

IBM supplies SAMPMJCL, a member of SYS1.SAMPLIB. The CSECT name in SAMPMJCL is MSTJCL05. SAMPMJCL contains the source statements for MSTJCL00. IBM provides this member so you can modify the master JCL according to your needs. Once you have modified the member, you must re-assemble and link edit it into the system.

Changes to Master JCL

Changes to the master JCL data set fall into three categories: modifying an existing statement, adding or deleting a statement, and using alternate versions of the data set.

Modifying a Statement: To modify a particular statement, use the AMASPZAP service aid program.

Adding and Deleting a Statement: To delete an existing statement or add a new one, make the change, reassemble the IEEMSJCL module, adding a CSECT card and an END card. Link edit it into SYS1.LINKLIB (or a library concatenated to LINKLIB via a LNKSTxx member of SYS1.PARMLIB).

Notes:

1. *If you add any DD statements to the module, make sure the data sets are created before the IPL that is to make use of them: if the allocation of any data set defined in IEEMSJCL fails, the IPL also fails.*
2. *For the primary job entry subsystem, do not include the START command in the master JCL and as a parameter in the SYS1.PARMLIB(IEFSSNxx). If you include the START command in the master JCL for the primary job entry subsystem, you must specify the NOSTART parameter for the primary job entry subsystem in the IEFSSNxx member of SYS1.PARMLIB*
3. *Until the primary job entry subsystem is started, no work can be done that requires JES input or output services.*

Alternate Versions of the Master JCL

You can select alternate versions of the master JCL data set at IPL using the MSTRJCL= system parameter. The parameter can be in an IEASYSxx member of SYS1.PARMLIB. See *MVS/XA SPL: Initialization and Tuning* for the syntax of the MSTRJCL= parameter and for details of creating IEASYSxx members of PARMLIB. *MVS/XA SPL: Initialization and Tuning* also includes the syntax for the IEFSSNxx member.

Chapter 9. Customizing the System Trace Table

Among the various tracing facilities available to MVS installations is the trace run by the system itself to record significant software events.

The system trace table appears in a formatted dump as shown in *MVS/XA Diagnostic Techniques*. System trace entries are normally formatted by SNAP or print dump (AMDPRDMP) processing as part of dump formatting.

In MVS/XA, you can customize the system trace table to include your own installation-defined events. For each event you want to trace, you put a trace table entry (TTE) in the system trace table, using the PTRACE macro. The PTRACE macro creates an explicit trace table entry identified as USRn, where “n” is a single hexadecimal digit from 0 to F. See “Using the PTRACE Macro.”

The MVS/XA system trace table provides for 16 user-defined explicit trace table entries. IBM supplies 16 identical routines to format the USRn TTEs; if you want to format a USRn entry differently from the way the default routine would format it, you can replace the IBM-supplied version with your own. See “Formatting a USRn Trace Table Entry.”

The USRn System Trace Table Entry

The unformatted user trace table entry is mapped by macro IHATTE, as documented in the *MVS/XA Debugging Handbook*. An explicit trace table entry of the USRn type includes a three-word header that identifies the entry type and indicates the length of the TTE and the time of day of the event being recorded. Following the header are:

- The current TCB address
- The home address space identifier (HASID)
- The primary address space identifier (PASID)
- The secondary address space identifier (SASID)
- The return address of the program issuing the PTRACE macro
- Continuation information
- User data, in up to five fullwords starting at TTEWRD5

When the TTE is printed in a dump, its contents are formatted in columns. See *MVS/XA Diagnostic Techniques* for examples of the formatted system trace table, and Figure 9-2 for an example of the formatted USRn TTE.

Using the PTRACE Macro

The PTRACE macro allows you to trace from one to 1024 fullwords of your own data as part of the system trace facility. Your user data can be the contents of a register or a range of registers, or a data area.

To issue the PTRACE macro, a program must be running in key 0, in supervisor state. The program can be in either 24- or 31-bit addressing mode.

The syntax for the PTRACE macro is:

```
PTRACE  TYPE=USRn, [REGS=(r1,r2)] ,SAVEAREA=STANDARD  
        [REG=(1)]
```

where

USRn

designates the type of entry and supplies the ID to be assigned to the TTE. The value you substitute for “n” identifies your entry to the system. It can range from X'0' to X'F'.

REGS

indicates the data you want traced.

REGS=(r1,r2): you can place up to 11 fullwords of data in a range of registers from 2 through 12, and code the register range on the REGS keyword. For example, REGS=(2,7) indicates that six fullwords of data, in registers two through seven, are to be traced.

REGS=(1): you can place the data in a parameter list and put the address of the list into register 1. Using the parameter list, you can specify up to 1024 fullwords of data to be recorded. The parameter list consists of a fullword indicating the number of words of data to be traced, followed by the data.

SAVEAREA = STANDARD

indicates the kind of save area being provided. It must be STANDARD, as shown.

See *SPL: System Macros and Facilities* for details on coding the PTRACE macro. Some additional programming considerations follow:

- If you want to associate your TTE with a particular processor, issue PTRACE while the processor is disabled for I/O and external interrupts.
- The PTRACE facility records trace data only when the system trace is active. System trace is activated at IPL and remains active until specifically deactivated by the TRACE operator command; it can run concurrently with the generalized trace facility (GTF).
- PTRACE treats all addresses passed to it as 31-bit addresses.

Figure 9-1 shows two examples of the PTRACE macro.

```
PTRACE TYPE=USR1,REGS=( 3 , 3 ) ,SAVEAREA=STANDARD  
PTRACE TYPE=USRB,REGS=( 1 ) ,SAVEAREA=STANDARD
```

Figure 9-1. Examples of the PTRACE Macro

The first example is a request to create a trace table entry for one fullword of user data, which is contained in register 3.

The second example is a request to trace the user data in a parameter list whose address is in register 1.

PTRACE Macro Processing

PTRACE processing creates one or more TTEs in the system trace table, assigning each a code corresponding to the acronym ID (USR0 - USRF) that you specify on the macro. PTRACE includes in the entries both system-required data, and some system-supplied data, to help you identify and use the TTE in a printed dump. Each USRn TTE can include up to five fullwords of data to be traced; if you request more than that, PTRACE creates additional TTEs that are continuations of the first one.

Multi-Part Trace Table Entries

If you code the PTRACE macro as follows:

```
PTRACE TYPE=USR3,REGS=( 2 , 12 ) ,SAVEAREA=STANDARD
```

you are requesting that 11 fullwords of data be traced. Each word of data is in a general purpose register. In this case, the single invocation of the PTRACE macro results in three TTEs being placed in the system trace table. All three are identified by the code corresponding to USR3; the first and second entries each contain 5 fullwords of user data, and the third contains 1 fullword of user data.

If the program in which you code the PTRACE macro can be interrupted, it is possible that the three TTEs are not placed consecutively in the system trace table. The continuation information included in multi-part entries allows you to find the related TTEs – and, thus, your user data – in the formatted dump.

Figure 9-2 shows the continuation information generated for the USR3 TTEs requested in the example above. In the formatted TTE, the continuation information is in the column headed UNIQUE-1.

...IDENT...ADDRESS	UNIQUE-1	UNIQUE-2	UNIQUE-3	...PASD	SASD	TIMESTAMP	RECORD
...USR3...81A007B6	003D 000	00000001	00000002	...0009	0009	93786447F3C83680	
	00000003	00000004	00000005				
...USR3...81A007B6	003D 014	00000006	00000007	...0009	0009	93786447F3C83B00	
	00000008	00000009	00000005				
...USR3...81A007B6	003D 028	0000000B		...0009	0009	93786448F3C8420	

Figure 9-2. Continuation Information from PTRACE for Multi-Part TTE

The first halfword of continuation information is a PTRACE identification count, the same for all the parts of a multi-part TTE. This count relates the parts of a multi-part TTE to each other.

The second halfword of continuation information contains the relative byte offset of the user data in a particular part. X'00' indicates the first byte of user data, X'14' indicates the twenty-first byte, and so on. You use the continuation information to locate all the pieces of a multi-part TTE in a formatted dump.

The user data in the example in Figure 9-2 is in the columns headed UNIQUE-2 through UNIQUE-6. The first part of the USR3 TTE contains the data in registers 2 through 6; the second part of the TTE contains the data in registers 7 through 11; and the third part of the TTE contains the data in register 12.

Formatting a USRn Trace Table Entry

IBM supplies 16 identical routines to format the USRn trace table entries. Their module names correspond to the USRn identifiers 0 through F: ITRF007F through ITRF0F7F.

Following is a brief explanation of the processing involved in formatting a USRn trace table entry using the IBM-supplied routines.

The system trace filter/formatter module, IEAVETEF, determines, from the home ASID in the TTE, if a particular trace table entry is selected for formatting. If so, IEAVETEF formats the following system-supplied status information for the TTE in the trace output buffer:

Note: The labels in parentheses are those mapped by macro IHATROB.

- The processor number (TROBPRID)
- The HASID (TROBASID)
- The current TCB address (TROBTCBA)
- The USRn acronym for the TTE (TROBID)
- The return address of the issuer of PTRACE (TROBRET)

- The continuation information, if the TTE is part of a multi-part entry (TROBUNQ1)
- The PASID (TROBPASN)
- The SASID (TROBSASN)
- The time-of-day value for the TTE (TROBTIME)

IEAVETEF then calls the appropriate USRn formatting routine (ITRF0n7F), which calls ITRFDEFU, the default formatting routine.

ITRFDEFU formats the user data, in hexadecimal, in the trace output buffer fields labelled TROBUNQ1, TROBUNQ2, and TROBUNQ3. These fields correspond to the columns headed by UNIQUE-1, UNIQUE-2, and UNIQUE-3 in the printed trace table. ITRFDEFU then calls the print buffer service routine (IEAVETPB) to print the output line. A single USRn TTE can contain up to five fullwords of user data, which are formatted on two successive lines in the printed output. ITRFDEFU calls the print buffer service routine once for each successive output line.

Replacing a USRn TTE Formatting Routine

You may replace any of the ITRF0n7F formatting routines with one that fits your installation's requirements, link editing it into load module IEAVETFC in SYS1.LPALIB.

Generally, your routine must conform to the same conventions and requirements the ITRF0n7F routines supplied by IBM follow. Specifically:

Parameters Passed to the USRn Formatter

When IEAVETEF passes control to a USRn formatting routine, register 1 contains the address of a parameter list. The parameter list contains:

1. Address of a **token**
2. Address of the **trace output buffer**
3. Address of the **USRn TTE**
4. Address of a **512-byte work area**
5. Address of a byte containing the **subpool number** to be used for additional work space

Each time the formatting routine calls the print buffer service routine, IEAVETPB, it must pass it the token, which was received as the first parameter.

The trace output buffer, the second parameter, is mapped by IHATROB. It is initialized with the status information and, if the TTE is a multi-part entry, with continuation information in TROBUNQ1, which corresponds to the UNIQUE-1 column of the formatted output line.

The USRn formatting routines must be reentrant. The work area received as the fourth parameter allows this. Your formatting routine may use the 512 bytes for any purpose; we recommend that it be used as an automatic data area. The same work area is passed to each USRn formatting routine; it is not zeroed out between calls. It is, however, initialized to zeroes before the first call to a USRn routine.

If your USRn routine needs more than 512 bytes for its processing, it can obtain more storage via the GETMAIN macro. The fifth parameter is the subpool that must be used for this GETMAIN request.

Note: If your routine does issue a GETMAIN, be sure that it also issues a FREEMAIN for that storage. If it does not free the storage reserved for its use when it is running on behalf of a SNAP dump request, the storage it reserves will remain allocated and unavailable for the life of the job.

Return Codes From the USRn Formatter

When your USRn formatting routine returns control to IEAVETEF, it must put a return code into register '15, based on whether or not it formatted the TTE. The possible codes and their meanings are:

Code	Meaning
0	The USRn TTE was formatted
4	The USRn TTE was not formatted

Printing the Trace Output Buffer Contents

Your routine must update the trace output buffer with the user data and call the print buffer service routine, IEAVETPB, to print each output line.

IEAVETPB, entry point in module IEAVETFA, prints the trace output buffer. In the case of a print dump request, IEAVETPB also keeps track of the number of lines printed on a page and skips to a new page when the maximum has been printed or the TTE being printed requires more than the lines left on the page.

When your USRn formatting routine passes control to IEAVETPB, register 1 must contain the address of a five-fullword parameter list. The parameter list contains:

1. Address of the token received from IEAVETEF
2. Address of the trace output buffer (TROB)
3. Address of the relative output line number
4. Address of the number of output lines expected
5. Address of the print option for this call

The trace output buffer print service routine uses the parameters as follows:

1. If it does not receive the token, it issues a X'09E' ABEND with a reason code of X'00005301', and the system trace formatter terminates.
2. It locates the output line to be printed using the address in the second parameter.

3. The third parameter is the relative output line number for the formatting of a single TTE. The value in this parameter indicates which line this is of the total number of output lines needed to format the TTE.
4. The fourth parameter is the total number of lines needed to format the TTE; that is, the number of times the USRn formatting routine will call IEAVETPB to print a particular TTE.

For a print dump request, IEAVETPB uses the number of output lines expected and the relative output line value to determine whether the entire TTE can fit on the same page. If the TTE cannot be formatted on the current page, IEAVETPB prints the entire TTE on the next page.

5. The fifth parameter indicates the kind of output contained in the trace output buffer:
 - X'80000000' means the output buffer contains a TTE
 - X'00000000' means the output buffer contains a message

Having printed a line of output, IEAVETPB returns to the USRn formatting routine with a return code of zero in register 15.

Handling Errors During TTE Formatting

If your USRn formatting routine encounters a program check, the ESTAE for IEAVETFC (the system trace formatter controller) gets control. The ESTAE tests the completion code. If it is X'0C6' through X'0CF', indicating a likely data-dependent arithmetic or conversion error, the following takes place:

- A message is printed in the trace table output saying that the USRn format routine failed and is disabled.
- The USRn TTE that caused the failure is formatted by ITRFDEFU, the default USRn formatter routine.
- Future USRn TTEs that would have been formatted by the failing routine will also be formatted by ITRDEFU.

If the completion code is other than X'0C6' through X'0CF', IEAVETFC terminates and a message is printed saying that the trace formatter failed because of an unrecoverable error.

Figure 9-3 is a sample of the assembler language code needed to format a USRn trace table entry. The sample CSECT formats a USR0 TTE that was created by the following PTRACE:

```
PTRACE TYPE=USR0,REGS=(2,4),SAVEAREA=STANDARD
```

```

ITRF007F CSECT ,
ITRF007F AMODE 31
ITRF007F RMODE ANY
*-----REGISTER EQUATES
R0      EQU 0
R1      EQU 1
R2      EQU 2
TTEPTR  EQU 7          TTE ADDRESSABILITY
R8      EQU 8          MODULE ADDRESSABILITY
R9      EQU 9          DATA AREA ADDRESSABILITY
TROBPTR EQU 12         TROB ADDRESSABILITY
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
*-----STANDARD ENTRY LINKAGE
      DS 0H
      USING *,R15          TEMPORARY MODULE ADDRESSABILITY
      DROP R15            DROP TEMPORARY ADDRESSABILITY
      STM R14,R12,12(R13) SAVE REGISTERS
      BALR R8,0
      USING *,R8          MODULE ADDRESSABILITY
*****
*      OBTAIN ADDRESSABILITY TO THE PARAMETERS *
*****
      L   R9,12(,R1)      ADDRESS OF THE AUTOMATIC DATA AREA
*                               IS THE FOURTH PARAMETER
      USING DATA,R9     DATA AREA ADDRESSABILITY
      ST  R13,SAVE0001+4  BACKWARD CHAIN SAVEAREAS
      LA  R2,SAVE001      ADDRESS OF MODULE SAVEAREA
      ST  R2,8(R13)      FORWARD CHAIN SAVEAREAS
      LR  R13,R2          POINT TO CURRENT SAVEAREA
*-----SAVE THE TOKEN
      L   R2,0(,R1)      ADDRESS OF THE TOKEN
      ST  R2,TOKEN       SAVE TOKEN IN AUTOMATIC AREA
*-----TRACE OUTPUT BUFFER ADDRESSABILITY
      L   TROBPTR,4(,R1)  GET TROB ADDRESS FROM PARMLIST
      USING TROB,TROBPTR TROB ADDRESSABILITY
*-----USRO TTE ADDRESSABILITY
      L   TTEPTR,8(,R1)  ADDRESS OF THE CURRENT TTE
      USING TTE,TTEPTR   SET USRO TTE ADDRESSABILITY

```

Figure 9-3 (Part 1 of 3). Sample Code for Formatting USRn Trace Table Entries

```

*****
*          FORMAT A HIGHLIGHTING MESSAGE          *
*****
L      R2,TOKEN          TOKEN TO BE PASSED
ST     R2,ETPBLIST      SET 1ST PARAMETER(TOKEN)
LA     R2,HILITE        120 CHAR MESSAGE TO BE OUTPUT
ST     R2,ETPBLIST+4    SET 2ND PARAMETER
LA     R2,LINE1         RELATIVE LINE NUMBER OF THE LINE
*
ST     R2,ETPBLIST+8    SET 3RD PARAMETER
LA     R2,MAXLINES      NUMBER OF LINES OF OUTPUT EXPECTED
ST     R2,ETPBLIST+12   SET 4TH PARAMETER
LA     R2,CPSMSG        THE OUTPUT IS A MESSAGE
ST     R2,ETPBLIST+16   SET 5TH PARAMETER
LA     R1,ETPBLIST      LOAD ADDRESS OF PARAMETER LIST
L      R15,IEAVETPB     LOAD ADDRESS OF IEAVETPB ROUTINE
BALR   R14,R15         CALL IEAVETPB
*****
*          INITIALIZE THE OUTPUT BUFFER WITH USRO DATA WORDS          *
*****
MVC    WORK5,TTEWRD5     MOVE USER WORD TO WORK AREA
UNPK   WORK10,WORK5      UNPACK USER DATA WORD 1
TR     WORK10,EBCTABL    TRANSLATE TO PRINTABLE HEX
MVC    TROBUNQ1,WORK10+1 MOVE TO OUTPUT BUFFER UNIQUE1 COLUMN
MVC    WORK5,TTEWRD6     MOVE USER WORD TO WORK AREA
UNPK   WORK10,WORK5      UNPACK USER DATA WORD 2
TR     WORK10,EBCTABL    TRANSLATE TO PRINTABLE HEX
MVC    TROBUNQ2,WORK10+1 MOVE TO OUTPUT BUFFER UNIQUE2 COLUMN
MVC    WORK5,TTEWRD7     MOVE USER WORD TO WORK AREA
UNPK   WORK10,WORK5      UNPACK USER DATA WORD 3
TR     WORK10,EBCTABL    TRANSLATE TO PRINTABLE HEX
MVC    TROBUNQ3,WORK10+1 MOVE TO OUTPUT BUFFER UNIQUE3 COLUMN
*****
*          FORMAT THE USRO TRACE TABLE ENTRY          *
*****
L      R2,TOKEN          TOKEN TO BE PASSED
ST     R2,ETPBLIST      SET 1ST PARAMETER(TOKEN)
LA     R2,TROB          TROB TO BE OUTPUT
ST     R2,ETPBLIST+4    SET 2ND PARAMETER
LA     R2,LINE1         RELATIVE LINE NUMBER OF THE LINE
*
ST     R2,ETPBLIST+8    SET 3RD PARAMETER
LA     R2,MAXLINES      NUMBER OF LINES OF OUTPUT EXPECTED
ST     R2,ETPBLIST+12   SET 4TH PARAMETER
LA     R2,CPTTE         THE OUTPUT IS A PART OF A TTE
ST     R2,ETPBLIST+16   SET 5TH PARAMETER
LA     R1,ETPBLIST      LOAD ADDRESS OF PARAMETER LIST
L      R15,IEAVETPB     LOAD ADDRESS OF IEAVETPB ROUTINE
BALR   R14,R15         CALL IEAVETPB

```

Figure 9-3 (Part 2 of 3). Sample Code for Formatting USRn Trace Table Entries

```

*****
*      RETURN TO THE CALLER WITH A RETURN CODE OF 0      *
*****
EXIT  LA R15,0          LOAD UP THE RETURN CODE
L     R13,4(R13)       LOCATE CALLERS SAVE AREA
L     R14,12(R13)      RESTORE THE RETURN ADDRESS
LM    R0,R12,20(R13)   RESTORE REGISTERS
BR    14

*-----CONSTANTS
EBCTABL DS    OD
        EQU    *          TRANSLATE TABLE FOR PRINTABLE HEX
        ORG    *+240
        DC    C'0123456789ABCDEF'
LINE1   DC    F'1'        RELATIVE LINE NUMBER 1 FOR HILITE
MAXLINES DC    F'1'      TOTAL NUMBER OF OUTPUT LINES
HILITE  DC    CL120'*****THE MUCH AWAITED USRO TRACE EVENT
        HAS OCCURRED! *****'

*-----
CPTTE   DC    X'80000000'  IEAVETPB OPTIONS WORD VALUE
*       THE OUTPUT IS A TTE
CPMSG   DC    X'01000000'  IEAVETPB OPTIONS WORD VALUE
*       THE OUTPUT IS A MESSAGE

*-----
IEAVETPB DC    V(IEAVETPB) ADDRESS OF IEAVETPB ROUTINE
*-----DYNAMIC DATA AREA
DATA    DSECT
SAVE0001 DS    18F        STANDARD SAVEAREA
ETPBLIST DS    5F        IEAVETPB PARAMETER LIST
TOKEN   DS    F          ADDRESS OF THE TOKEN TO BE PASSED
*       TO IEAVETPB
WORK5   DS    CL5        INPUT WORK AREA FOR USER DATA
        DS    0F
WORK10  DS    CL10       OUTPUT WORK AREA FOR USER DATA
ENDDATA EQU    *
ITRF007F CSECT
SIZDATA DC    AL4(((ENDDATA-DATA+7)/8)*8)
        IHATROB
        IHATTE USRN=YES
        END

```

Figure 9-3 (Part 3 of 3). Sample Code for Formatting USRn Trace Table Entries

Fitting your Subsystems into the System

In order to work with MVS or MVS/XA, all subsystems must be defined to the operating system, by name. During IPL, you must define the primary job entry subsystem (JES) to MVS, so you can use the JES input and output services. During IPL, you also define any secondary subsystems.

Once a subsystem is defined to the operating system, MVS provides some services that help you use it efficiently. This part of *System Modifications* includes discussions of the process of defining a subsystem to MVS and of one of the services available for subsystems, the subsystem affinity service.

The following books are mentioned in this part of *System Modifications*:

- *MVS/XA SPL: Initialization and Tuning*
- *MVS/XA MVS Configuration Program Guide and Reference*

Chapter 10. Defining Subsystems To the Operating System

To specify your **primary job entry subsystem** and **secondary subsystems** to the operating system, you use the IEFSSNxx member of SYS1.PARMLIB. The IEFSSNxx members contain subsystem names and, optionally, the names of corresponding subsystem initialization routines and parameters for input to the initialization routines. With the PRIMARY parameter, you specify the primary job entry subsystem to use. The NOSTART parameter indicates that an automatic start for the primary job entry subsystem should **not** be issued. NOSTART may be specified in conjunction with the PRIMARY parameter and is not available for use with the secondary subsystem specifications.

For each subsystem you define, MVS constructs a subsystem communication vector table (SSCVT). MVS processes subsystems defined in IEFSSNxx members of SYS1.PARMLIB, in the order specified with the SSN system parameter. For information on specifying the SSN system parameter, see *MVS/XA SPL: Initialization and Tuning*.

During master scheduler initialization, the subsystem initialization routines specified in the IEFSSNxx member receive control as part of the master scheduler initialization process.

Defining Subsystems in Members of SYS1.PARMLIB

At system initialization, you can indicate the specific IEFSSNxx SYS1.PARMLIB members needed for this IPL by specifying the SSN= system parameter in

1. the IEASYSxx member of SYS1.PARMLIB or
2. through the operator's response to the "specify system parameters" message.

Notes:

1. *IEFSSN00 is the system default parmlib member and contains the definition for the default primary job entry subsystem (JES2).*
2. *If you do not specify the SSN parameter for an IPL, the master scheduler looks for IEFSSN00; if it exists, the subsystems specified in IEFSSN00 are defined to MVS.*

Defining subsystems in IEFSSNxx members of parmlib is flexible:

- You can change the parmlib members each time you IPL the system, without having to do a reassembly and link edit.
- You can include parameters in IEFSSNxx to be passed to a subsystem initialization routine.

Each SYS1.PARMLIB entry must be on a single record, and only one entry per record is allowed. There can be only one subsystem per record. The format of the IEFSSNxx SYS1.PARMLIB member is shown in Figure 10-1.

<pre>ssname [,init-routine [,parm][,PRIMARY][,NOSTART]] comments</pre>
ssname A 1- to 4-character name defining the subsystem. The first character must be either alphabetic or national and the remaining characters must be alphameric or national.
init-routine A 1- to 8-character name corresponding to the entry point of the initialization routine for the associated subsystem.
parm A variable-length field containing data that is passed to the initialization routine.
PRIMARY This parameter indicates that the specified subsystem name (ssname) will be the primary subsystem name.
NOSTART This parameter indicates that an automatic start for the primary subsystem is not to be issued.
comments Begin comments with a blank following the other fields of the record.

Figure 10-1. Format of the SYS1.PARMLIB member, IEFSSNxx.

Notes:

1. The `SYS1.PARMLIB` record need not begin in column 1.
2. When `ssname` terminates with a blank as the delimiter instead of a comma, the system assumes no initialization routine is present. Any remaining data on the record is ignored.
3. If you specify parameters without specifying an initialization routine, the parameters are ignored.
4. If blanks, commas, or apostrophes are included in the parameter field, the entire field must be enclosed in apostrophes. If you want an apostrophe to be treated as part of the parameter, code two consecutive apostrophes.
5. If you want to specify a primary subsystem but you do not want to specify an initialization routine or any parameters, you must use commas to replace the positional parameters. For example, if you want `JES2` as the primary subsystem but no initialization routine or parameters, you would code `JES2,,,PRIMARY`

Passing Parameters to the Initialization Routine

The master scheduler initialization routine issues a LINK to the subsystem initialization routines specified in the `IEFSSNxx` member of `SYS1.PARMLIB`.

On input to the initialization routine, register 1 points to eight bytes of storage. The first word points to the `SSCVT` for that subsystem and the second word points to a 12-byte parameter list. Macro `IEFJSIPL`, residing in `SYS1.MACLIB`, formats the parameter list as follows:

Offset	Field Name	Contents
0	JSILGTH	Length of parameter list
1	JSICONID	Console id to which the subsystem initialization routine is to issue messages (see note 1)
2	JSILGTPR	Length of user parameters (see notes 2 and 3)
3	JSIRSV0	Reserved
4	JSIADRPR	Address of user parameters that are specified in <code>SYS1.PARMLIB</code> record (see note 3)
8	JSIRSV1	Reserved

Notes:

1. When `IEFJSINT` or `IEFJSIN2` calls `IEFJSBLD` to build the parameter list, the `JSICONID` field is set to zeroes. These are the default for the master console.
2. Apostrophes that enclose a parameter are not used in the parameter length. Also, when an apostrophe is part of the parameter and you code two apostrophes, only one is used in determining the parameter length. For example, if `'PARM'S'` appears in the record, it appears as `PARM'S` (with a length of 6 bytes) to the subsystem initialization routine.
3. `MVS` places zeros in parameter fields `JSILGTPR` and `JSIADRPR` for a subsystem if the `IEFSSNxx` record does not include parameters.

Figure 10-2. Parameter List for Subsystem Initialization Routines

Register 13 points to an 18-word save area.

The subsystem initialization routine receives control in supervisor state, key 0.

It is the responsibility of the subsystem initialization routine to establish its own recovery and perform all cleanup on its behalf.

System Handling of Duplicate Subsystems

You should not define a subsystem to MVS more than once in an IPL.

If the system detects a duplicate name after it has built an SSCVT for a subsystem, it will not give control to any initialization routine associated with the duplicate name. Rather, it issues message IEE730I: "DUPLICATE SUBSYSTEM NOT INITIALIZED."

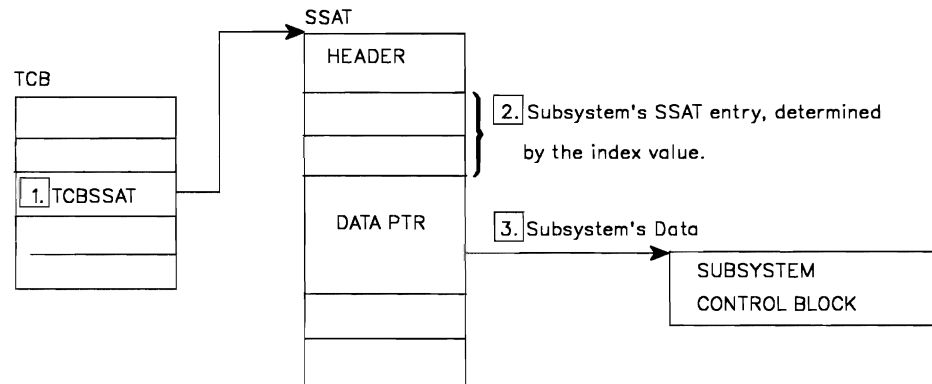
There are two exceptions to this rule:

1. If the name of a primary job entry subsystem was encountered previously and it matches the name of the current subsystem, the system checks the SSCVT for the status of its initialization. If the subsystem has not been initialized, MVS give the initialization routine control during master scheduler initialization.
2. If the name of a subsystem was encountered previously and the same name is now specified as PRIMARY, the following processing occurs:
 - a. The SSCVT that was created for the first subsystem name remains on the SSCVT chain. (This is to maintain a record of the subsystem specification.)
 - b. A new SSCVT is created for the PRIMARY specification and it becomes the first SSCVT on the chain.

Chapter 11. The Subsystem Affinity Service

The TCB subsystem affinity service maintains and manages a subsystem affinity table (SSAT), which consists of one-word entries for every subsystem running in an IPL. A subsystem's SSAT entry can be used to hold a pointer to its own control blocks or data areas, thus removing its dependence on information passed by the problem program.

Each valid TCB has an associated SSAT. Figure 11-1 shows how the SSAT field in the TCB can point eventually to a subsystem's control blocks.



Notes:

1. Address of the SSAT. MVS creates a unique SSAT when processing the first SSAFF SET request for a particular TCB, and stores its address in the TCBSSAT field.
2. Each subsystem has a unique entry in a specific TCB's SSAT. Each entry consists of one word. In this figure, the entry is a pointer to the subsystem's control blocks.
3. The subsystem first places a value in the SSAT by issuing the SSAFF SET request. The subsystem obtains the value from the SSAT, during subsequent execution, by issuing a SSAFF OBTAIN request. In this figure, the value allows the subsystem to reference its own control blocks.

Figure 11-1. Subsystem Affinity Service

The subsystem acquires its SSAT index value by invoking the VERIFY SUBSYSTEM function through the IEFSSREQ macro. See “IEFSSREQ: Obtaining The SSAT Index Value”

MVS assigns each entry in an SSAT to a unique subsystem based on the subsystem's index value. The subsystem can use its SSAT entry for each TCB under which it runs.

An SSAFF SET request places one fullword of subsystem-passed data in the SSAT identified by the TCB keyword. This allows the subsystem to put its entry in the SSAT of any TCB to which it has addressability.

An SSAFF OBTAIN request extracts and returns to the subsystem the fullword of data from the SSAT entry identified by the current TCB and the subsystem's SSAT index value. The OBTAIN request works only for the SSAT pointed to by the current TCB.

Note: A subsystem that uses the TCB subsystem affinity service cannot rely on information stored in an SSAT over a checkpoint/restart: the SSAT value could change from one system initialization to another. For additional information about the restrictions and use of the checkpoint/restart facility, see *MVS Checkpoint/Restart*.

SSAFF: Set/Obtain Subsystem Affinity

You can use the SSAFF macro to SET or OBTAIN an entry in a subsystem affinity table (SSAT).

Before you issue the SSAFF macro, register 13 must point to an 18-word save area.

The format of the SSAFF macro is:

[symbol]	SSAFF	{ SET [,TCB=tcb-address] OBTAIN ,DATA=data-address ,ENTRY=index-value }
----------	-------	----------------------------------------------------------------------------------

One blank is required before and after "SSAFF."

SET requests have the following requirements:

- The caller must be enabled and in supervisor state, key 0.
- The caller must not be in cross-memory mode.
- The TCB whose SSAT will be set must be in the caller's home address space and must be either the current TCB or a subtask of the current TCB. If any of these conditions are not satisfied, the calling routine abends.

OBTAIN requests have the following requirements:

- The caller must be in task mode. If this condition is not met, the calling routine abends.
- The caller must have current addressability to the home address space. If this condition is not met, the results are unpredictable.

The SSAFF macro parameters have the following meanings:

symbol

any valid assembler language symbol.

SET

indicates that MVS is to place the value specified by the DATA parameter into the subsystem's associated SSAT entry. The SET request destroys the contents of registers 14, 15, 0, 1, and 2.

OBTAIN

indicates that MVS is to place the contents of the specified SSAT entry of the issuing task in the register or data area specified by the DATA parameter. The OBTAIN request destroys the contents of registers 14, 15, 0, and 1.

,TCB = tcb-address — RX-Type Address, or Register (2)-(12)

this parameter, valid only for SET requests, specifies the register or storage location that contains the address of the TCB whose SSAT MVS is to use when processing the SET request.

Note: If you omit the TCB keyword, MVS uses the current task's TCB. If you allow this default, the calling program must include the IHAPSA mapping macro to identify the current TCB.

,DATA = data-address — RX-Type Address, or Register (1) or (3)-(12)

For SET, this parameter specifies the register or fullword storage location that contains the subsystem's data. MVS stores the data in the SSAT for a SET request.

For OBTAIN, this parameter specifies the register or fullword storage location that is to contain the value extracted from the SSAT.

MVS places a value of zero in the data area if any one of the following is true during an OBTAIN request:

- The SSAT entry associated with the specified index-value contains a zero.
- A null SSAT exists for the caller. (A SET request was not performed prior to the OBTAIN request.)
- The specified index value exceeds the size of the caller's SSAT.

,ENTRY = index-value — RX-Type Address, or Register (0) or (3)-(12)

this parameter specifies the register or fullword storage location that contains the subsystem's SSAT index value. If you specify an index value that is greater than the number of currently-defined subsystems (subsystems defined using the methods described in Chapter 10, "Defining Subsystems To the Operating System"), the request fails.

For SSAFF SET requests, the subsystem affinity service uses:

- The TCB address to locate the required SSAT. When the subsystem does not supply the TCB address, MVS uses the currently-executing TCB (PSATOLD).
- The SSAT index value to locate the specific SSAT entry that is to be set.

For SSAFF OBTAIN requests, the subsystem affinity service uses:

- The currently-executing TCB to locate the required SSAT.
- The SSAT index value to locate the specific SSAT entry to be returned.

IEFSSREQ: Obtaining The SSAT Index Value

You obtain the SSAT index value by using the IEFSSREQ macro to invoke the VERIFY SUBSYSTEM function.

The IEFSSREQ macro has the following requirements:

- Register 13 must point to an 18-word save area.
- You must include the CVT and IEFJESCT mapping macros in the calling program.
- The caller can be in TCB or SRB mode.

The syntax of the IEFSSREQ macro is:

[symbol]	IEFSSREQ
----------	----------

symbol

any valid assembler language symbol.

One blank is required before and after “IEFSSREQ.”

When the VERIFY SUBSYSTEM function is invoked, Register 1 points to a fullword of storage which points to the subsystem options block (SSOB). The caller must supply some information in the SSOB, as indicated below. *Macro IEFSSOBH*, residing in SYS1.MACLIB, formats the subsystem options block (SSOB) as follows:

SSOBID	Identifier, 'SSOB'
SSOBLEN	Length of SSOB header
SSOBFUNC	Function ID. For a verify subsystem request, this field contains SSOBVERS (a field defined in IEFSSVS).
SSOBSSIB	Pointer to the subsystem information block (SSIB).
SSOBRETN	Return code field
SSOBINDV	Pointer to the verify subsystem extension (IEFSSVS)

Macro IEFSSVS, residing in SYS1.MACLIB, formats the verify subsystem extension (IEFSSVS) as follows:

SSVSLEN Length of the extension

All other fields must be zero.

Macro IEFJSSIB, residing in SYS1.MACLIB, formats the subsystem information block (SSIB) as follows:

SSIBID Identifier, 'SSIB'

SSIBLEN Length of SSIB

SSIBSSNM The name of the master subsystem ('MSTR'). (This field might be altered by verify subsystem requests. If more than one request is being made, 'MSTR' must be stored here before each call.)

SSIBJBID Subsystem name to be verified. This is an 8-byte field. The subsystem name must be left-justified and padded with blanks.

The VERIFY SUBSYSTEM function places a return code in register 15. The caller should examine the return code to determine if the request was processed. The possible return codes are:

Code	Meaning
0	The verify subsystem request was processed
4	The subsystem specified in SSIBSSNM does not support the VERIFY SUBSYSTEM function
8	The subsystem specified in SSIBSSNM exists but is not active
12	The subsystem specified in SSIBSSNM is not defined to MVS
16	The pointer to the SSOB is invalid
20	The SSOB or SSIB have invalid lengths or formats

If the request was processed (Register 15=0), the return code in the SSOB (SSOBRETN) indicates whether the subsystem name in the SSIB (SSIBJBID) is valid. Return codes in SSOBRETN and their meanings are:

Code	Meaning
0	Valid subsystem name
4	Invalid subsystem name

Upon verification of a valid subsystem name, the verify subsystem extension will contain:

SSVSSCTP Pointer to the subsystem's SSCVT

SSVSNUM The SSAT index value that you use in a SSAFF macro request.

Note: Prior to using this value as the SSAT index value in the ENTRY parameter of the SSAFF macro, you must place the value in a register or in storage on a fullword boundary.

If the caller of the verify subsystem function fails to provide a verify subsystem extension (IEFSSVS), the address of the SSCVT and the SSAT index value will not be available to the subsystem.

Notes:

1. *The subsystem index value is valid only for use on the MVS processor on which it was obtained, and during the current IPL.*
2. *A valid SSAT index value is returned only for subsystems defined via the methods described in "Defining Subsystems to MVS"*

Cross Memory Considerations: The control blocks you access in using the subsystem affinity service (SSOB, SSIB, and verify subsystem extension) must be addressable from the address space in which the verify subsystem request is made.

For further information on the use of cross memory services, see *MVS/XA SPL: System Macros and Facilities*.

MVS/XA System Services

Whenever you add features to your installation's system, you give up some of the system's performance efficiency. MVS provides various system services that can help minimize this drop in performance, and can help you maintain or modify your system as needed. The services include verification services, input/output facilities, and performance aids.

The following books are mentioned in this part of *System Modifications*:

- *IBM 3800 Printing Subsystem Programmer's Guide*
- *MVS Interactive Problem Control System (IPCS) User's Guide and Reference*
- *MVS/XA Checkpoint/Restart*
- *MVS/XA Data Management Services*
- *MVS/XA Data Management Macro Instructions*
- *MVS/XA Debugging Handbook*
- *MVS/XA Diagnostic Techniques*
- *MVS/XA JCL*
- *MVS/XA JES3 Diagnosis*
- *MVS/XA Operations: System Commands*
- *MVS/XA Message Library: System Messages*
- *MVS/XA SPL: Initialization and Tuning*
- *MVS/XA SPL: JES2 Initialization and Tuning*
- *MVS/XA SPL: JES3 Initialization and Tuning*
- *MVS/XA SPL: Service Aids*
- *MVS/XA SPL: Supervisor Services and Macro Instructions*
- *MVS/XA SPL: System Macros and Facilities*
- *MVS/XA SPL: 31-Bit Addressing*



Chapter 12. Unit Verification Service

The scheduler provides the unit verification service. The unit verification service enables you to obtain information from the eligible device table (EDT) and to check your device specification against the information in the EDT.

There are two versions of the unit verification service:

1. IEFEB4UV, for problem programs or authorized callers, and
2. IEFGB4UV or IEFAB4UV, for authorized callers.

Unit Verification Service

The unit verification service performs the following functions:

- check groups
- check units
- return unit name
- return UCB addresses
- return group ID
- return look-up value
- convert device type to look-up value
- return attributes
- indicate unit name is a Look-up value
- check units with no validity bit
- specify subpool for returned storage
- return unit names for a device class

Check Groups

This function determines whether the input device numbers make a valid allocation group. To be valid, the device grouping must include either all the device numbers being verified, or none of them. If this is not the case, the allocation group is split and the input device numbers do not make up a valid allocation group.

Check Units

This function determines whether the input device numbers correspond to the unit name in the EDT. In addition to a return code in register 15, it sets to one the high-order flag bit of any invalid device numbers in the parameter list.

Return Unit Name

This function returns the unit name associated with a look-up value provided as input. The unit name is the EBCDIC representation of the IBM generic device type (for example, 2305) or the esoteric group name (for example, TAPE) from the EDT.

A look-up value is an internal representation of the unit name, used as an index into the EDT. See “The Eligible Device Table” for more information about the EDT. Because teleprocessing devices do not have generic device names, you cannot use this function to request information about teleprocessing devices.

Return UCB Addresses

This function returns the UCB pointer list associated with the unit name provided as input.

Return Group ID

This function returns the allocation group ID corresponding to each UCB address specified in the input list.

Return Look-up Value

This function returns the four-byte internal representation of the unit name that serves as an index into the EDT. It is the converse of the return unit name function.

Convert Device Type to Look-up Value

This function will convert a four-byte UCB device type to an internal representation of the unit name, to serve as an index into the EDT. The convert device type to look-up value function allows programs that have only a four-byte UCB device type to query the EDT. It may be used whenever a look-up value is required as input to the unit verification service.

Return Attributes

This function returns general information about the specified unit name.

Indicate Unit Name is a Look-up Value

The input to the check units and return UCB addresses functions can be specified as a four-byte internal representation of the unit name rather than as the unit name itself.

Check Units With No Validity Bit

This function causes the check units function to set only a return code in case of an invalid device number. The no validity bit function saves processing in the case of a check units request for a single unit: because the parameter list is not modified, it need not be in key 1 storage. The function is available through IEFGB4UV or IEFAB4UV only.

Specify Subpool for Returned Storage

This function is used with the return UCB addresses function or with the return unit names for a device class function. It allows you to specify a particular subpool to return the requested information in. This function is available through IEFEB4UV only.

Return Unit Names for a Device Class

This function returns a list of IBM generic device types (for example, 2305) and/or esoteric group names (for example, TAPE) associated with the input device class. This function is available through IEFEB4UV only.

Callers of IEFEB4UV

The unit verification routine, IEFEB4UV, is for both problem program callers and for authorized callers. It runs in task mode in the caller's key.

To use IEFEB4UV, the calling program must do the following:

- Create the input data structures and parameter list
- Place the address of an 18-word save area in register 13
- Provide a recovery environment
- Pass control to IEFEB4UV using the LINK macro

On return, IEFEB4UV restores all registers except register 15, which contains a return code.

Callers of IEFGB4UV or IEFAB4UV

The IEFGB4UV and IEFAB4UV routines are for authorized callers and run in task mode in scheduler key (1). To use the IEFGB4UV or IEFAB4UV unit verification routine, the calling program must do the following:

- Create the input data structures and parameter list in non-fetch protected key 1 storage
- Place the address of an 18-word save area in register 13
- Provide a recovery environment if authorized
- Pass control to IEFGB4UV or IEFAB4UV using the LINK macro

On return, the routine restores all registers except register 15, which contains a return code.

IEFAB4UV is the 24-bit addressing mode interface that is provided for compatibility with previous releases. It changes the environment to 31-bit addressing mode and calls IEFGB4UV to perform the requested functions. On return, it converts the environment back to 24-bit addressing mode before returning to the caller.

Input To and Output From Unit Verification Service Routines

You must supply a two-word parameter list when invoking one of the unit verification routines (IEFGB4UV, IEFAB4UV, or IEFEB4UV).

The first word contains the address of a unit table. The contents vary according to the function(s) requested.

The second word contains the address of a two-byte field (FLAGS), in which you specify the function(s) requested.

The bits in the FLAGS parameter field have the following meanings:

Bit	Function Requested
0	Check groups
1	Check units
2	Return unit name
3	Return UCB addresses
4	Return group ID
5	Indicate unit name is a look-up value
6	Return look-up value
7	Convert device name to a look-up value
8	Return attributes
9	Check units with no validity bit
10	Specify subpool for returned storage
11	Return unit names for a device class
12-15	Reserved

Input Parameter List

The following diagram shows the input parameter list needed to invoke any unit verification service routine.

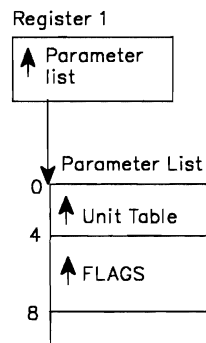


Figure 12-1. Input Parameter List

Input and Output Data Structures

The diagrams on the following pages show the input data structures and parameters needed to invoke each unit verification service routine. The output data structure returned by the routines is also shown.

You must declare the structures exactly as shown to get the response indicated by the function(s) you request in **FLAGS**.

Because many of the input and output data structures are the same, you can request many of the functions in combinations with other functions. The following table lists the valid single functions and combinations of functions that you can request in a single invocation of the unit verification service.

Code	Verification Service
0	
0,1	
0,1,5	
0,1,5,9	IEFGB4UV or IEFAB4UV only
1	
1,5	
1,9	IEFGB4UV or IEFAB4UV only
1,5,9	IEFGB4UV or IEFAB4UV only
2	
2,7	
2,8	
2,7,8	
3	
3,5	
3,8	
3,10	IEFEB4UV only
3,5,7	
3,5,10	IEFEB4UV only
3,8,10	IEFEB4UV only
3,5,7,10	IEFEB4UV only
4	
6	
6,8	
7	
8	
10,11	IEFEB4UV only
11	IEFEB4UV only

Register 15 if Request Fails: On return, register 15 will contain a return code. If the invocation fails, it may be for one of the following reasons:

1. If you request an invalid function or an invalid combination of functions, register 15 contains a return code of 28 and the request fails.
2. If the JES control table (JESCT) does not contain a valid pointer to the EDT, the environment is incorrect. Register 15 contains a return code of 24. The request fails.

Requesting Function Code 0 (Check Groups)

The check groups function is available through any of the unit verification services.

Input: Set bit 0 in FLAGS to 1.

The input unit table structure is shown below.

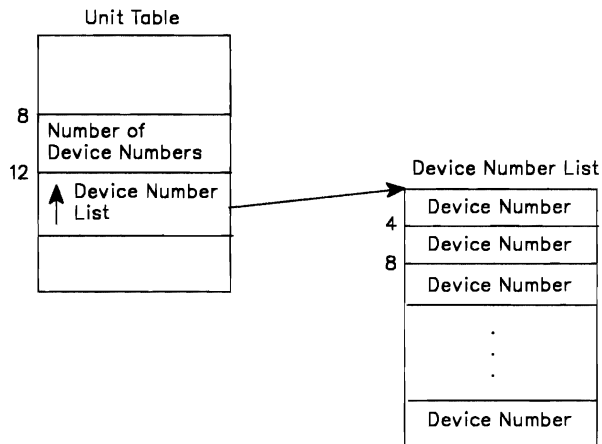


Figure 12-2. Requesting Function Code 0 (Check Groups)

Output: None.

Register 15 contains one of the following return codes:

Code	Meaning
0	The specified input is correct.
12	The device groupings are invalid.

Requesting Function Code 1 (Check Units)

The check units function is available through any of the unit verification services.

Input: Set bit 1 in **FLAGS** to 1.

The input unit table structure is shown below. The device number list with the **FLAG** byte initialized to 0 for each device number.

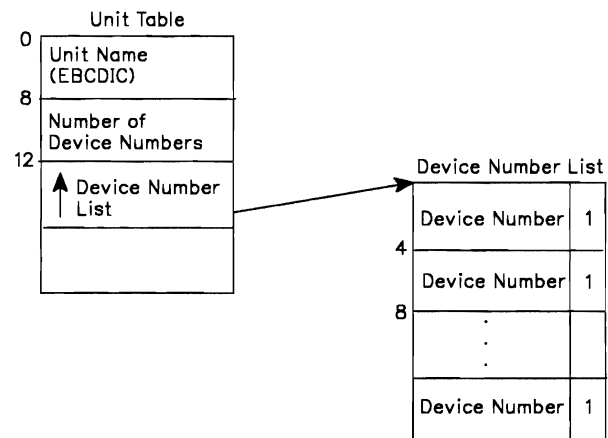


Figure 12-3. Requesting Function Code 1 (Check Units)

Output: If the device number is invalid, bit 0 of **FLAG** byte is set to 1.

Register 15 contains one of the following return codes:

Code	Meaning
0	The specified input is correct.
4	The specified unit name is invalid.
8	Unit name has incorrect units assigned.
20	One or more device numbers are invalid.

Requesting Function Code 2 (Return Unit Name)

The return unit name function is available through any of the unit verification services.

Input: Set bit 2 in FLAGS to 1.

The input unit table structure is shown below.

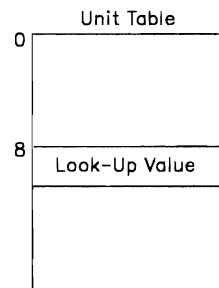


Figure 12-4. Requesting Function Code 2 (Return Unit Name)

Output: The unit table contains the unit name as shown in the following figure.

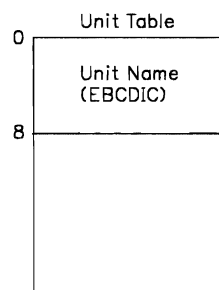


Figure 12-5. Output from Function Code 2 (Return Unit Name)

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit table contains the EBCDIC unit name.
4	The look-up value could not be found in the EDT.

Requesting Function Code 3 (Return UCB Addresses)

The return UCB addresses function is available through any of the unit verification services.

Input: Set bit 3 in **FLAGS** to 1.

The input unit table structure is shown below.

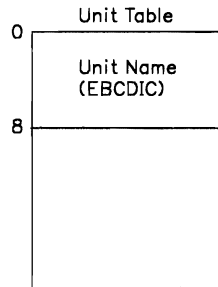


Figure 12-6. Requesting Function Code 3 (Return UCB Addresses)

Output: The unit table contains a pointer to the UCB Pointer List as shown in the following figure.

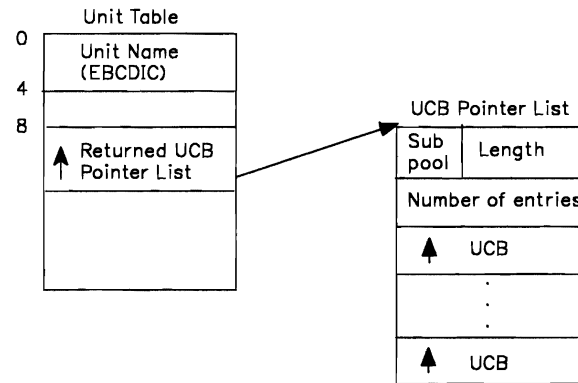


Figure 12-7. Output from Function Code 3 (Return UCB Addresses)

For authorized callers, the list is returned in the default subpool, 230. For unauthorized callers, the subpool default is 0. See function code 10 for a description of how to change the default subpool. The caller must free the number of bytes in the length field from the subpool before exiting.

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit table contains the pointer to the UCB pointer list.
4	The unit name could not be found in the EDT.
16	Storage was not available for the UCB pointer list.

Requesting Function Code 4 (Return Group ID)

The return group id function is available through any of the unit verification services.

Input: Set bit 4 in FLAGS to 1.

The input unit table structure is shown below.

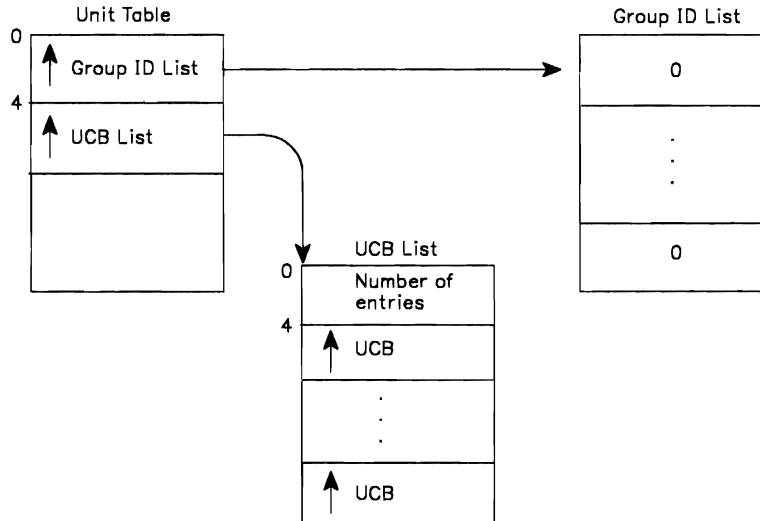


Figure 12-8. Requesting Function Code 4 (Return Group ID)

Note: There is one fullword in the group id list for each UCB in the UCB list.

Output: The group id list contains the group id corresponding to each UCB in the input UCB list.

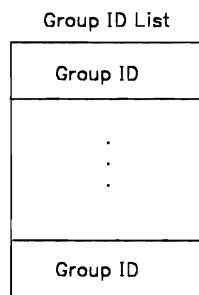


Figure 12-9. Output from Function Code 4 (Return Group ID)

Note: If the UCB is not in the EDT, the group id for that particular entry remains zero.

Register 15 contains a 0.

Requesting Function Code 5 (Indicate Unit Name is a Look-up Value)

The indicate unit name is a look-up value function is available through any of the unit verification services.

Input: Set bit 5 in FLAGS to 1.

The input unit table structure is shown below.

This function is not valid by itself. It must be used in combination with other functions that require an input unit name. If you know the look-up value corresponding to the unit name, you can substitute it for the unit name in the input unit table. The following figure represents the first two fullwords of the unit table when function code 5 is requested.

Unit Table

0	Look-up Value
4	0

Figure 12-10. Requesting Function Code 5 (Indicate Unit Name is a Look-up Value)

Output: None specifically associated with this function.

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
4	The input look-up value could not be found in the EDT.

Requesting Function Code 6 (Return Look-up Value)

The return look-up value function is available through any of the unit verification services.

Input: Set bit 6 in FLAGS to 1.

The input unit table structure is shown below.

This function is the opposite of the return unit name function (Code 2). The following figure represents the unit table structure when you request function code 6.

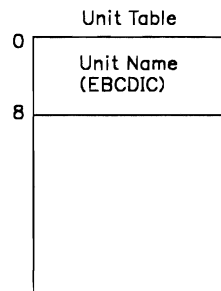


Figure 12-11. Requesting Function Code 6 (Return Look-up Value)

Output: The unit table contains the look-up value.

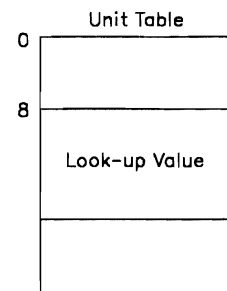


Figure 12-12. Output from Function Code 6 (Return Look-up Value)

Register 15 contains one of the following return codes:

Code	Meaning
0	Processing is successful.
4	The unit name could not be found; no look-up value is returned.

Requesting Function Code 7 (Convert Device Type to Look-up Value)

The convert device type to look-up value function is available through any of the unit verification services.

Input: Set bit 7 in FLAGS to 1.

The input unit table structure is shown below.

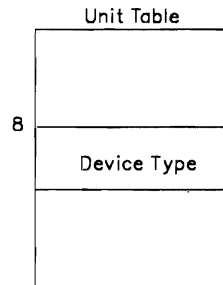


Figure 12-13. Requesting Function Code 7 (Convert Device Type to Look-up Value)

Note: The device type is in the format of the UCBTYP field of the UCB.

Output: The unit table contains the look-up value.

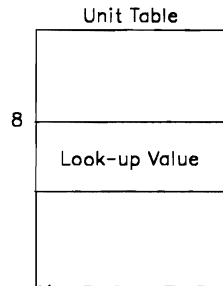


Figure 12-14. Output from Function Code 7 (Convert Device Type to Look-up Value)

The conversion of the device type to a look-up value is done in place. There is no error checking of the device type.

Register 15 contains a zero.

Requesting Function Code 8 (Return Attributes)

The return attributes function is available through any of the unit verification services.

Input: Set bit 8 in **FLAGS** to 1.

The input unit table structure is shown below.

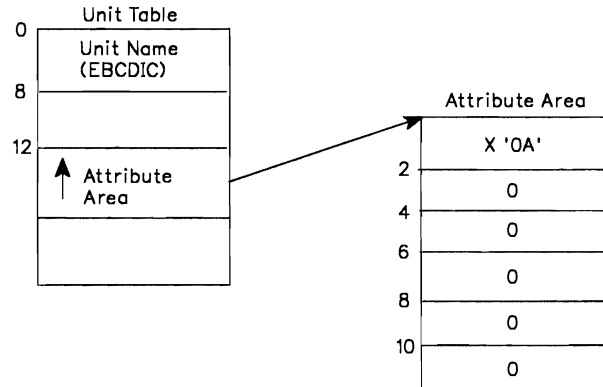


Figure 12-15. Requesting Function Code 8 (Return Attributes)

Output: The attribute area contains the following:

Byte	Contents
0	Length of the attribute area (X'0A') This must be filled in prior to calling the unit verification service.
1-2	Flags describing the unit name: <ul style="list-style-type: none"> ● Bit 0 on — unit name is an esoteric group name ● Bit 1 on — unit name is VIO-eligible ● Bit 2 on — unit name contains 3330V units ● Bit 3 on — unit name contains TP class devices ● Bits 4-7 are not used.
3	Number of device classes in the unit name
4-7	Number of generic device types in the unit name
8-9	Reserved

Register 15 contains one of the following return codes:

Code	Meaning
0	The unit name was found; the attributes are returned.
4	The unit name was not found; no attributes are returned.

Requesting Function Code 9 (Check Units with No Validity Bit)

The check unit with no validity bit function is available through IEFAB4UV and IEFGB4UV only.

Input: Set bit 9 in FLAGS to 1.

There is no other input associated with this function except the input for the check units function. This function must be used in combination with the check units function (code 1).

Output: See the output from the check units function.

The FLAG byte of the device number list is not altered; only the return code from the check units function (code 1) is available to determine if any device numbers in the device numbers list are invalid.

Requesting Function Code 10 (Specify Subpool for Returned Storage)

The specify subpool for returned storage function is available only through IEFEB4UV.

Input: Set bit 10 in FLAGS to 1. This function is not valid alone and must be used with either the return UCB addresses function (code 3) or the return unit name function for a device class (code 11). The input unit table structure is shown in the following figure.

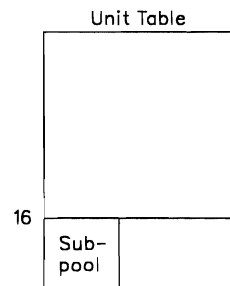


Figure 12-16. Requesting Function Code 10 (Specify Subpool for Returned Storage)

Output: See the output from the function that this is invoked in combination with.

The subpool field of the returned list contains the input subpool, and the returned list resides in that subpool. No error checking of the subpool is performed. If the subpool is invalid, the unit verification routine fails.

Requesting Function Code 11 (Return Unit Names for a Device Class)

The return unit names for a device class function is available only through IEFEB4UV.

Input: Set bit 11 in FLAGS to 1.

The following figure shows the input unit table structure.

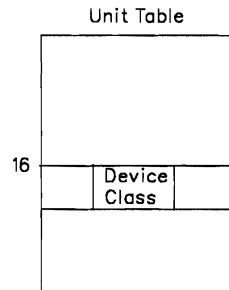


Figure 12-17. Requesting Function Code 11 (Return Unit Names for a Device Class)

Output: The unit table contains the pointer to the names list as shown in the following figure.

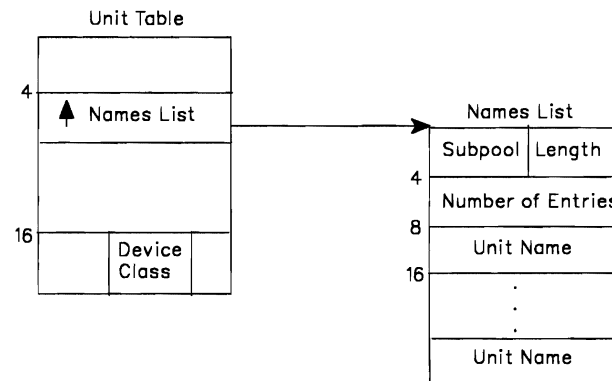


Figure 12-18. Output from Function Code 11 (Return Unit Names for a Device Class)

For authorized callers, the subpool that the names list is returned in is the default subpool 230. For unauthorized callers, the default subpool is 0. To change this default, see the description for function code 10 (specify subpool for returned storage). The caller must free the number of bytes in the length field from the subpool before exiting.

Register 15 contains one of the following return codes:

Code	Meaning
0	The pointer to the names list is stored in the unit table.
16	Storage was not available for the names list.

Requesting Multiple Functions

The following examples show the input to and output from multiple functions.

Example 1 shows the multiple functions of codes 0 and 1.

Example 2 shows the multiple functions of codes 3 and 10.

Example 3 shows the multiple functions of codes 1 and 5.

Example 1 Function Codes 0 and 1

Input

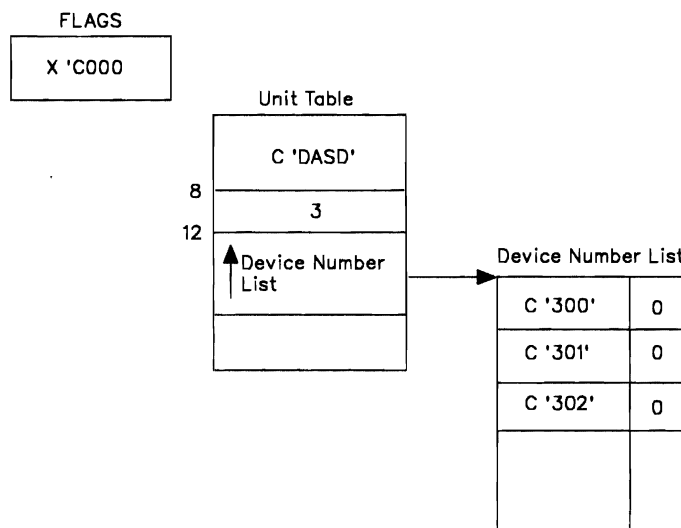


Figure 12-19. Input for Function Codes 0 and 1

Example 1 Function Codes 0 and 1 (continued)

Output

Device Number List

C '300'	0
C '301'	0
C '302'	0

Register 15

0

Figure 12-20. Output from Function Codes 0 and 1

Note: All input device numbers make up a single allocation group and are associated with the esoteric unit name DASD.

Example 2 Function Codes 3 and 10

Input

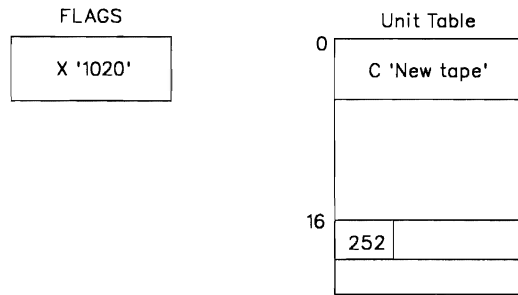


Figure 12-21. Input for Function Codes 3 and 10

Output

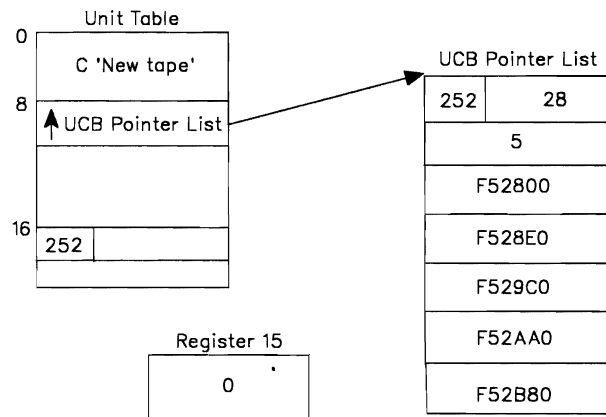


Figure 12-22. Output from Function Codes 3 and 10

Note: The caller must be authorized to request subpool 252. The unit verification service invoked must be IEFEB4UV because function code 10 is requested. The caller must free the UCB pointer list before exiting.

Example 3 Function Codes 1 and 5

Input

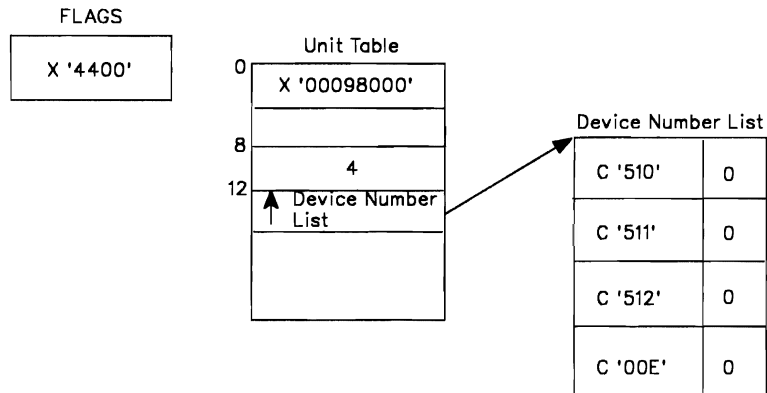


Figure 12-23. Input for Function Codes 1 and 5

Output

Device Number List

C '510'	0
C '511'	0
C '512'	0
C '00E'	X '80'

Figure 12-24. Output from Function Codes 1 and 5

Note: Device 00E did not belong to the unit name that was associated with the input look-up value.

Chapter 13. The Hot I/O Detection Table

Some I/O devices and control units generate unsolicited interrupts in the normal course of their operation. A hardware malfunction that causes repeated, unsolicited I/O interrupts is called “hot I/O.” Undetected, such hot I/O interrupts can cause the system to loop, or to use up the system queue area. The I/O supervisor (IOS) attempts to detect a hot I/O condition and perform recovery before the system requires a re-IPL.

One of the tasks of the installation system programmer is to define the threshold that the I/O supervisor is to use to detect hot I/O. You may also specify the recovery actions IOS is to take. For either task, you use the IECIOSxx member of SYS1.PARMLIB.

When the number of repeated unsolicited I/O interrupts exceeds the installation-defined threshold value, the system assumes there is a hot I/O condition.

When the system detects a hot I/O condition, it checks the recovery action it is to take according to your specifications on the HOTIO parameters in IECIOSxx. If you specify the OPER option on the HOTIO parameters in IECIOSxx, the system requests operator intervention.

IOSRHIDT: The HIDT

Module IOSRHIDT resides in the nucleus and contains the hot I/O detection table. The IOSDHIDT macro (in SYS1.AMODGEN) maps the HIDT. See the *MVS/XA Debugging Handbook*.

The HIDT contains the *device threshold value* that determines how many consecutive unsolicited interrupts must occur before a given device is considered “hot.” The threshold value can range from 0 to 32,767 (X'7FFF'). A value of 0 indicates that no hot I/O detection is to be done. Too low a threshold value could result in false hot I/O detection.

The HIDT also contains codes that indicate default *recovery actions*, along with anchors to the control blocks used in determining and processing hot I/O conditions.

The recovery actions are assigned codes according to whether the hot device is a reserved DASD, a nonreserved DASD, or any other device.

The valid recovery action codes are:

Code	Meaning
X'00'	Operator is to specify the recovery action. This code causes IOS to issue message IOS110A, IOS111A, or IOS112A, depending upon the device involved.
X'02'	Box the hot device (force it offline).
X'04'	Initiate channel path recovery on the channel path over which the last interrupt was received.
X'05'	Force the channel path offline.

Figure 13-1. Valid Hot I/O Recovery Action Codes

Any other values are invalid. When the system encounters an invalid recovery action option, it issues a message (see “Modifying the HIDT”) and obtains the recovery action from the operator.

The system “obtains recovery action from the operator” by issuing a message to the console requesting operator intervention. If the HIDT value indicates an action code other than X'00', the system performs the action; it also issues message IOS109I to tell the operator that it is performing hot I/O recovery.

Default Values in IOSRHIDT

The version of IOSRHIDT supplied by IBM includes preset values for the variables in the HIDT. To change these default values, use the options on the internal parameters for HOTIO as defined in the IECIOSxx member of SYS1.PARMLIB. See *MVS/XA SPL: Initialization and Tuning*. Figure 13-2 shows the default values and their meanings.

Offset	Name	Meaning ¹	Default Value ²
24 (X'18')	HIDDTHR	Hot I/O Threshold	X'64' (decimal 100)
28 (X'1C')	HID110DN	Hot non-DASD device; hot condition is not recursive.	X'02'
29 (X'1D')	HID110DR	Hot non-DASD device; hot condition is recursive.	X'02'
30 (X'1E')	HID111DN	Hot non-reserved DASD; hot condition is not recursive.	X'04'
31 (X'1F')	HID111DR	Hot non-reserved DASD; hot condition is recursive.	X'02'
32 (X'20')	HID112DN	Hot reserved DASD; hot condition is not recursive.	X'04'
33 (X'21')	HID112DR	Hot reserved DASD; hot condition is recursive.	X'00'

Notes:

¹ *Recursive* – If hot I/O recovery has been performed for a device, and that device becomes 'hot' again without accepting a start I/O request, the hot condition is called 'recursive'. If a recursive hot condition persists (three or more occurrences on a device), the system ignores any nonzero recovery code in the HIDE and requests operator intervention.

² See Figure 13-1 for the meanings of the valid recovery action codes.

³ The recovery action defaults are identified with the messages generated by the system for the various types of hot I/O conditions:

- Message IOS110A indicates a hot condition for a device other than a DASD.
- Message IOS111A indicates a hot condition for a non-reserved DASD.
- Message IOS112A indicates a hot condition for a reserved DASD.

Figure 13-2. IBM Default Hot I/O Threshold and Recovery Actions.

Modifying the HIDE

To modify the threshold and code fields in IOSRHIDE, you use the IECIOSxx member of SYS1.PARMLIB. Include in it the appropriate default values for hot I/O for a given IPL.

MVS/XA Initialization and Tuning describes the format of a hot I/O record in IECIOSxx, and contains more information on using members of SYS1.PARMLIB for customization.



Chapter 14. The Internal Reader Facility

The internal reader is a software substitute for a card punch and a card reader, a tape drive, or a TSO terminal. Instead of putting a job into the system (via JES) on punched cards, or via tape, you can use the *output* of one job or step as the *input* to another job, which will be processed directly by the JES input service.

The internal reader facility is useful for several kinds of applications:

- You can use it to generate another job or a series of jobs from an already-executing job. An online application program may submit another job to produce a report, for example, so it does not have to do it itself.
- A job that produces a series of jobs can put its output to an internal reader for immediate execution. For example, a job that updates data bases and starts other applications based upon some input parameters or real-time events, can use the internal reader for its output.
- The operator can start utility programs to read jobs from disk or tape files and submit them to the system. The IBM-supplied procedure 'RDR' is an example of a program that does this (See *MVS/XA SPL: JES2 Initialization and Tuning*).
- The operating system itself uses internal readers for submitting the JCL to start up started tasks (STCINRDR) or TSO logons (TSOINRDR).
- A TSO logon session is, in effect, an executing program communicating with a terminal. When the TSO user submits a job, the internal reader is the mechanism used to perform the submittal.

Following is a discussion of the batch job internal reader, which is the facility you can use to submit a job from within another job.

Setting Up and Using an Internal Reader

The process of setting up and using an internal reader involves five tasks:

- Creating and allocating a data set
- Opening the data set
- Putting records into the data set
- Closing/deallocating the data set
- Passing the data set/records to the job entry subsystem for processing

Most of these tasks come under the heading of data management; they act as an interface to the primary job entry subsystem, which does the actual I/O.

Note: The VSAM interface and all data management macros and routines reside below the 16Mb line and operate in 24-bit addressing mode. Therefore, your internal reader data sets will be allocated storage below the 16Mb line. See *MVS/XA SPL: 31-Bit Addressing* for more information.

Allocating the Internal Reader Data Set

You allocate an internal reader data set, in any address space, either with JCL or dynamically, as follows:

- **Define the data set in the JCL for a job:**

```
//JOB JCL DD SYSOUT=(A,INTRDR)
```

Notes:

1. *“INTRDR” is an IBM-reserved name identifying the internal reader as the program to process this data set after it is created and written to.*
 2. *The SYSOUT class on this DD statement becomes the message class for the submitted job unless you specify MSGCLASS on the JOB statement.*
- **Use the following SVC 99 text unit keys to dynamically allocate an internal reader data set:**
 - DALSYSOU — define the SYSOUT data set and its class.
 - DALSPGNM — specify the SYSOUT program name (INTRDR).
 - DALCLOSE — request that INTRDR be deallocated at close.
 - DALRTDDN — request the return of the ddname assigned by dynamic allocation.

See *MVS/XA SPL: System Macros and Facilities* for the format details of dynamic allocation text unit keys.

Notes:

1. *The INTRDR data set has its own spool space; that is, other spool data sets for the creating job/task do not share the same collection of spool tracks. This data set isolation makes the INTRDR data set portable; it can be transferred from one address space to another, freeing the spool space for reuse when the JES has finished entering the job(s) from the data set into the system.*
2. *An INTRDR data set can contain any number of jobs.*
3. *The output destination of the INTRDR data set becomes the job origin of all jobs contained within it.*
4. *INTRDR data sets contain sequential, fixed-length records.*

Opening the Internal Reader Data Set

When JES2 or JES3 uses the internal reader, MVS data management routines open the data set for output. You can do the same thing by coding the OPEN macro, specifying the DSORG as PS. See *MVS/XA Data Management Services* for information on using the OPEN macro.

Opening the INTRDR data set identifies it to the primary JES and prepares it to receive records.

Sending Job Output to the Internal Reader

Code a WRITE (BSAM) or PUT (QSAM) macro to send records to the internal reader.

You can put the output from a group of jobs into the INTRDR data set using a JCL procedure; or you can enter them one job at a time.

You can select particular jobs for the internal reader, generate special job streams for the internal reader using MVS utilities, or allow the operator to submit production job streams to the internal reader from the console.

You can put the output from a secondary subsystem directly into the input stream for JES processing, via an internal reader data set.

Closing the Internal Reader Data Set

While your program is issuing PUT macros to write records to the internal reader data set, the internal reader facility is writing them into a buffer in your address space. When that buffer is full, the internal reader data set is closed. The contents of the buffer are copied by JES into another buffer in the JES address space (JES3 then spools the data), from which JES input processing can enter them into the system.

This input processing is the key to the internal reader: the records in the buffer are input to the job you have specified, although they started out as output records from another job. Instead of being sent to the JES output processor, they are read into the system by input processing.

Filling the buffer in your address space results in the records being sent to the JES for processing. Close processing terminates the data set; the JES considers it a completed job stream and treats it as input. You can also send an internal reader data set to the JES for processing by coding one of the following:

1. Code /*EOF as the last record in the job.

This JES2 control statement simulates the CLOSE instruction; it delimits the current job and makes it eligible for immediate processing by the JES2 input service.

If JES3 is the primary subsystem, /*EOF is a request for special end-of-record processing. The internal reader facility closes the data set without deallocating it, so it is still available for more records, and sends the job to the JES input service.

2. Code `/*DEL` as the last record in the job.

This JES2 control statement cancels the job, and requests the output from the job. The job is immediately scheduled for output processing. The output will consist of any JCL submitted so far, followed by a message indicating that the job has been deleted before execution.

If JES3 is the primary subsystem, `/*DEL` is treated like `/*EOF`. The data set is closed, but will be reopened when another record is written to it.

3. Code `/*PURGE` as the last record in the job.

This control statement is used only by JES2 internal readers. It cancels the current job and schedules it for purge processing; no output is generated for the job.

4. Code `/*SCAN` as the last record in the job.

This statement also applies only to JES2 internal readers. It requests that the current job be scanned for JCL errors, but not executed.

5. Close the internal reader data set, using the `CLOSE` macro, if you want it deallocated.

You can put several groups of output records into the internal reader data set simply by starting each group with another JCL `JOB` statement. The following example illustrates this.

```
//JOBA      JOB      D58ELM1,MORRIS
//GENER     EXEC     PGM=IEBGENER
//SYSIN     DD       DUMMY
//SYSPRINT  DD       SYSOUT=A,DEST=2NDFLOOR
//SYSUT2    DD       SYSOUT=(M,INTRDR)
//SYSUT1    DD       DATA

//JOB      JOB      D58ELM1,MORRIS,MSGLEVEL=(1,1)
//REPORT1   EXEC     PGM=SUMMARY
//OUTPUT    DD       SYSOUT=*
//INPUT     DD       DSN=REPORTA,DISP=OLD

//JOB      JOB      D58ELM1,MORRIS,MSGLEVEL=(1,1)
//REPORT2   EXEC     PGM=SUMMARY
//OUTPUT    DD       SYSOUT=A,DEST=3RDFLOOR
//INPUT     DD       DSN=REPORTB,DISP=OLD

/*EOF
```

The IBM-supplied utility program `IEBGENER` is executed by job A. It reads from `SYSUT1`, and submits to the internal reader, jobs B and C, which are report-producing programs. Note that the message class for jobs B and C will be `M`, the `SYSOUT` class on the internal reader `DD` statement. Also, the `OUTPUT` data set from job B, because it specifies `"*"` (defaulting to the job's message class), will be class `M`.

The `/*EOF` control statement following the JCL indicates that the preceding jobs can be sent immediately to the job entry subsystem for input processing. Coding the `CLOSE` macro would have the same effect.

See *MVS/XA SPL: JES2 Initialization and Tuning* and *MVS/XA SPL: JES3 Initialization and Tuning* for more information about setting up and using internal readers.

Requesting a Started Task To Execute on a Secondary Subsystem

You can request that MVS start a task under a specific secondary subsystem by using the START command. MVS will assign an internal reader to the secondary subsystem.

Not all subsystems can read JCL. Therefore, to route the JCL for a started task to a secondary subsystem, the subsystem must:

- be active when the START command is issued.
- support an internal reader.
- support job selection.

In an MVS/XA installation, JES2, JES3, and the Master Subsystem (MSTR) meet these conditions.

The SUB= keyword of the START command directs the JCL you specify to the internal reader of the subsystem you specify. For example, the command:

```
START ANYPROC ,SUB=JESX
```

sends the JCL procedure, ANYPROC, for the started task to the input service of the subsystem, JESX. JESX assigns a job number, selects the task for execution, and controls the task's output.

Restrictions when Routing the JCL to the Master Subsystem

There are four restrictions when you route the JCL procedure to the Master Subsystem.

1. The JCL procedure must not use DD SYSOUT.

The Master Subsystem cannot start any jobs requiring system output services. Therefore, if the JCL for a job includes the SYSOUT keyword on a DD statement, MSTR cannot start the job. For example, if you enter:

```
START RMF ,SUB=MSTR
```

the RMF task will start. However, if it attempts to open a SYSOUT data set, it will abend. The error message will indicate that there is a JCL error.

2. The JCL procedure may only allocate datasets in the master catalog.

You cannot use private catalogs when starting a task under the Master Subsystem.

3. The JCL procedure must specify a non-zero value on the TIME keyword on the EXEC statement.

If you do not specify a value for TIME, the task will abend. The message error will indicate that there is a TIME error.

4. The JCL procedure must reside in SYS1.PROCLIB.



Defaults For The Subsystem

If you do not code a `SUB=` parameter on the `START` command, the starting subsystem is the primary job entry subsystem. However, if the task to be started is a subsystem, the Master Subsystem starts the task.





Chapter 15. The External Writer

An external writer is a group of modules that perform output processing for data sets not eligible for processing by the primary job entry subsystem. An example of such a data set would be one destined for a non-IBM printer; or one to be stored on a device not supported by MVS.

IBM provides an external writer, as part of MVS, that has the following features:

- It runs as a started task, in its own address space, in 24-bit addressing mode.
- It removes data sets from the JES spool, dynamically allocates them, reads them, writes them to output devices, and dynamically deallocates them.
- It processes only those data sets that meet its selection criteria. You set these at IPL, but you can modify them via the START command.

The usual technique for setting data set selection criteria is to build a list of eligible SYSOUT classes for the devices that will use the external writer.

However, an external writer can access data sets according to any or all of the following characteristics:

- output class
 - job ID
 - forms specification
 - destination (LOCAL, or remote workstation name)
 - the name of your output writing routine
- You can use the external writer as provided by IBM, or you can rewrite parts of it to fit your own requirements. It was designed to be modified by its users.
 - As shipped, it is set up to write class-A SYSOUT data sets to local tape devices, but it can be used to write to any QSAM-supported device.

The two major parts of the IBM external writer that can be modified or replaced are the data set processing subtask (the output writing routine) and the output separator routine. In order to see how you could write your own versions of these modules, you need to know how IBM's versions work.

STDWTR: IBM Standard Output Writing Routine

In IBM's external writer, the output writing routine:

1. Issues an OPEN (J type) for the input data set previously taken from the JES spool.
 - The “input data set” started out as an output data set whose SYSOUT class made it eligible for processing by the external writer.
 - The output writing routine provides its own SYNAD error – handling routine, on behalf of both the input and output data sets. See *MVS/XA Data Management Macro Instructions* for more information about OPEN-J and SYNAD.
2. Reads the input data set, using the locate mode of the GET macro.
3. Calls a subroutine to handle ANSI and machine control-character differences and to handle conversions between the input records and the output data set.
4. Calls a routine to write records to the output data set, using the locate mode of the PUT macro.
5. Closes the input data set after it has been read, and returns control to the main logic control module of the external writer, using the RETURN macro and setting a return code.

The module name for the output writing routine is IEFSD087. If you alter or replace the output writing routine code, you must re-linkedit the new module with module IASXWR00, the external writer initialization routine. You must call the new module IEFSD087. IASXWR00 is executed by the IBM-supplied cataloged procedure, XWTR. See “The External Writer Cataloged Procedure.”

Parameter List for the External Writer

The main function of module IASXWR00 is to initialize a 12-byte parameter list (PARLIST) for the use of the other external writer routines.

The parameter list contains information about the output device and DCB addresses for each data set. Its format follows:

Byte 0	Bits 0-3 The three high-order bits describe the type of output device:
011. 2540 punch unit
001. 1403, 3203-5, 3211, or 3800 printer device
010. tape device with punch-destined output
000. tape device with printer-destined output
If bit 2 is on, the output unit is either a printer or a punch.	
Bytes 1-3	Not used, but must be present
Bytes 4-7	The address of the DCB for the opened output data set, where the external writer will put the input records.
Bytes 8-11	The address of the DCB for the input data set, from which the writer will obtain logical records. (When this parameter list is given to the external writer, the input data set is not open.)

Figure 15-1. External Writer Parameter List

The switches indicated by the three high-order bit settings in byte 0 can be used in translating control character information from the input records to the form required by the output device.

Replacing the Standard Output Writing Routine

Figure 15-2 shows the general logic flow of the standard external writer. The following paragraphs present more detailed requirements and programming considerations for coding a replacement for the standard output writing routine.

Writing Your Own Output Writing Routine

IBM's external writer includes the standard output writing routine, called STDWTR.

If, when the external writer is started from the console, the operator does not specify the name of your output writing routine as one of the data set selection criteria, STDWTR does the output processing for that data set.

The output writing routine must issue an OPEN-J macro to open the desired input data set. The processing program that invokes the external writer has opened, written to, and closed the input data set on a direct access device.

The data set to which the output writing routine will write records is opened before the routine is loaded. The IBM-supplied output writing routine writes type 6 records to the output data set, providing accounting support for SMF.

Coding Conventions for the Output Writing Routine

In order for your output writing routine to work in the external writer, the following conventions must be observed:

- The routine must be reentrant.
- You must link-edit it into the load module IASXWR00, and it must reside in SYS1.LINKLIB or a library concatenated to LINKLIB via a LNKLSTxx member of SYS1.PARMLIB.
- You must specify its name in the SYSOUT parameter of the appropriate output DD statement, in the JCL for the job or step that calls the external writer. For example, when you want the external writer to use IBM's output writing routine, you use IEFSD087, the module name.
- The routine must use standard entry and exit linkage, saving and restoring its caller's registers.

At entry, register 1 points to the parameter list (See Figure 15-1), and register 13 points to an 18-word save area.

- The routine must use the GETMAIN and FREEMAIN macros to acquire and release any necessary storage.
- The routine must return control to its caller via the RETURN macro, in the same addressing mode it was called in. It must put a return code in register 15:
 - A return code of 0 indicates that the data set was processed successfully to the output device.
 - A return code of 8 indicates that the routine was unable to process the data set because of an output error.

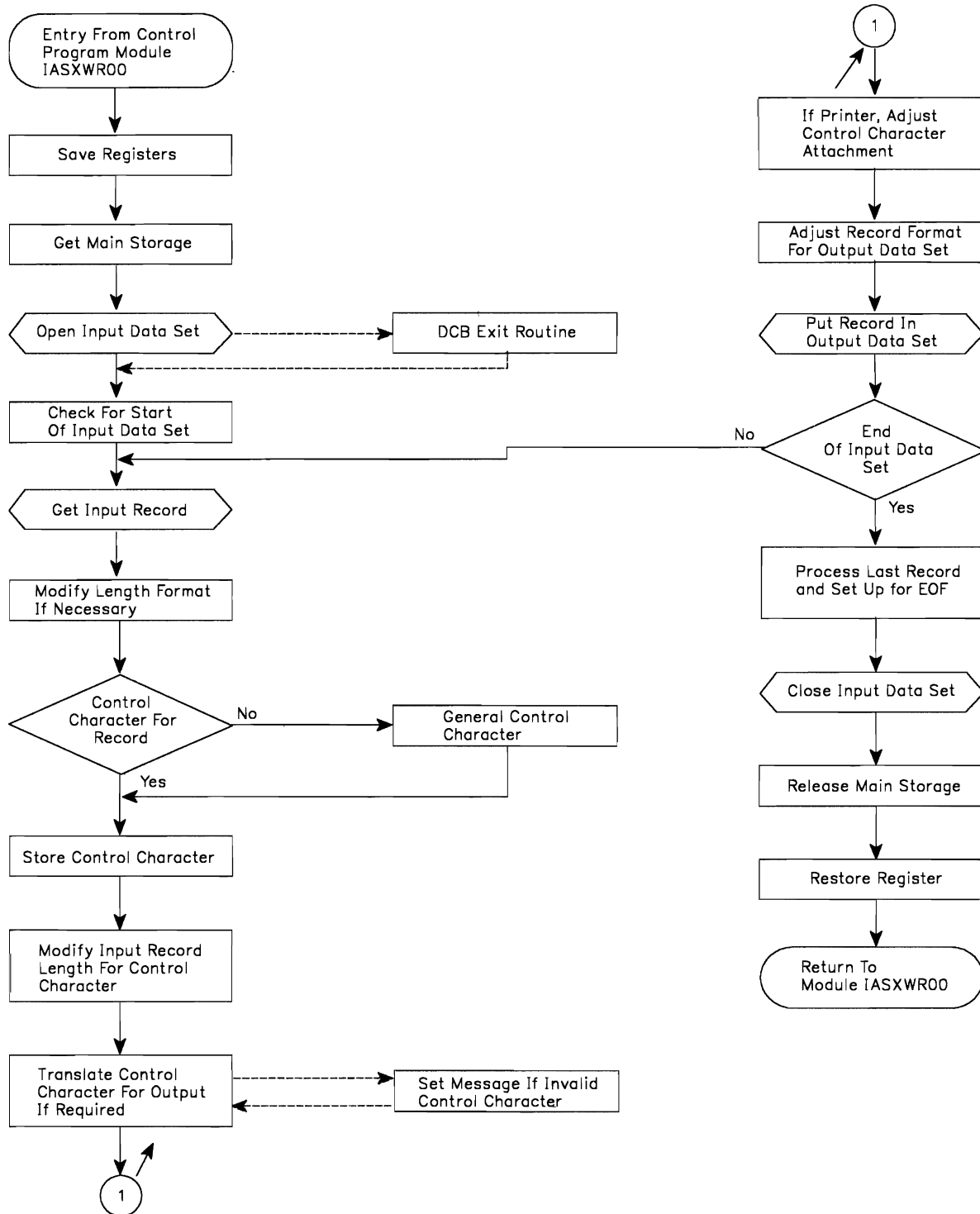


Figure 15-2. General Logic of IBM's External Writer Routine

Programming Considerations for the Output Writing Routine

In addition to the coding conventions, consider the following when writing your own output writing routine:

- **Obtaining Storage for Work Areas:** Using the GETMAIN macro, the output writing routine should obtain storage in which to set up switches and save record lengths and control characters.
- **Processing an Input Data Set:** To process a data set, the writing routine must get each record individually from the input data set, transform (if necessary) the record format and the control characters to conform to the output data set's requirements, and put the record in the output data set. Consider each of these tasks individually:

1. What must be done before the routine actually obtains a record from an input data set?

If the output device is a printer, the routine must provide a way to handle the two forms of record control character that are allowed in an output data set. Most printers are designed so that if the output data set records contain machine control characters, a record (line) is printed before the effect of its control character is considered. However, if ANSI control characters are used in the output records, the control character's effect is considered before the printer prints the line.

Thus, if the input data sets do not all have the same type of control characters, you will need to avoid overprinting the last line of one data set with the first line of the next.

When the input records have machine control characters and the output records are to have ANSI control characters, the standard (IBM-supplied) output writing routine produces a control character that indicates one line should be skipped before printing the first line of output data.

When the input records have ANSI control characters and the output records are to have machine control characters, the standard writing routine prints a line of blanks before printing the first actual output data set record. Following this line of blanks, the printer generates a one-line space before printing the first record.

Depending upon the characteristics of the printers in your installation, you will probably want your output writer to perform some kind of "printer initialization" like that outlined here.

2. After the output writing routine has properly opened the input data set, and has completed any necessary printer initialization, it must obtain records from the input data set.

The standard output writing routine uses the locate mode of the GET macro. If you use this macro, you will need to check the MACRF field of the input data set's DCB to see if GET in locate mode is allowed. If not, you can override the MACRF parameter on the GET macro itself.

See *MVS/XA Data Management Macro Instructions* for information on coding and using all the QSAM macros.

3. Having obtained a record from the input data set, the output writing routine must now make sure that its format and control character are compatible with the requirements of the output data set.

Because the output data set is already opened when the output writing routine is entered, your routine will have to adhere to the established conventions.

The standard output writing routine uses the PUT macro in the locate mode to write records to the output data set. For fixed-length output, it obtains the record length for the output data set from the DCBLRECL field of the DCB.

If an input record is longer than the length specified for the output records, the standard output writing routine truncates the input record from the right.

If an input record is shorter than the length specified for the output records, the standard output writer left-justifies the input record and pads the field with blanks.

When the output record length is variable and the input record length is fixed, the standard output writer adds control character information (if necessary) and variable record control information to the output record. Control character information is one byte, and record control information is four bytes long. Both additions are at the high-order end of the record.

If the output record is not at least 18 bytes long, the standard output writer pads it on the right with blanks.

The standard output writer also adjusts the length of the output record to match the length of the output buffer.

4. When the output writing routine has successfully adjusted the input and output records, it can read the input data set until end of data. At that point, you need to consider another aspect of input data set processing: what is going to happen to the last input record?

The standard output writer handles output to either card punch or printer, as required; your routine could also send output to an intermediate tape or DASD device. Depending upon the kind of device, the last few records obtained from the input data set will receive different treatment.

It might happen that all the records from a given data set are not available on the output device until the output of records from the next data set is started, or until the output data set is closed. When the output data set is closed, the standard output writer automatically puts out the last record of its last input data set.

For Punch Output: When a card punch is the output device, the last three output cards could still be in the machine when the input data set is closed.

To put out these three records with the rest of the data set, and with no breaks, the standard output writer provides for three blank records following the actual data set records.

For Printer Output: When a printer is the output device, the last record of the input data set is not normally put in the output data set at the time the input data set is closed.

To force out this last record, the standard output writer generates a blank record to follow the last record of the actual data set.

- **Closing Input Data Set(s):** After the standard output writer finishes putting out the records of an input data set, it closes the data set before returning control to the calling module. All input data sets must be closed.
- **Releasing Main Storage:** The output writing routine should release the storage it acquired, using the FREEMAIN macro, before returning to its caller.
- **Handling Errors:** The routine must put a return code into register 15 before returning to its caller using the RETURN macro.

The standard output writer sets a return code of 8 if it terminates because of an unrecoverable error on the output data set. Otherwise, the return code is 0: the output writing routine must handle input errors itself.

IEFSD094: The Output Separator Routine

Any output processing to a punch or printer must include a means of separating one job from another within the continuous deck or listing.

Part of IBM's external writer is module IEFSD094, which writes separation records to the output data set prior to the writing of each job's output.

You can modify this separator routine to suit your installation's needs, or you can create your own routine. IBM's version does the following:

- **For Punch-Destined Output:** The separator routine provides three specially-punched cards (deposited in stacker 1) prior to the punch card output of each job. Each of these cards is punched in the following format:

Columns	1 to 35	blanks
Columns	36 to 43	jobname
Columns	44 to 45	blanks
Column	46	output classname
Columns	47 to 80	blanks

- **For Printer-Destined Output:** The IBM-supplied separator routine provides three specially-printed pages prior to printing the output of each job. Each of these separator pages is printed in the following format:
 - Beginning at the channel 1 location (normally near the top of the page), the jobname is printed in block characters over 12 consecutive lines. The first block character of the 8-character jobname begins in column 11. Each block character is separated by 2 blank columns.
 - The next two lines are blank.
 - The output classname is printed in block characters covering the next 12 lines. This is a 1-character name, and the block character begins in column 35.
 - The remaining lines, to the bottom of the page, are blank.
 - In addition to the block characters, a full line of asterisks (*) is printed twice (that is, overprinted) across the folds of the paper. These lines are printed on the fold *preceding* each of the three separator pages, and on the fold *following* the third page. This is to make it easy for the operator to separate the job output in a stack of printed pages.

To control the location of the lines of asterisks on the page, the IBM-supplied separator routine requires that a channel 9 punch be included (along with the channel 1 punch) on the carriage control tape or in the forms control buffer (FCB). The channel 9 punch should correspond to the bottom of the page. The printer registration should be offset to print the line of asterisks on the fold of the page.

The IBM-supplied separator routine makes no provision for the 3800 printing subsystem; if you use it on a 3800, the FCB must locate a channel 9 punch at least one-half inch from the paper perforation.

Separator Routine Parameter List

IBM's external writer provides its separator routine with a 4-word parameter list of necessary information. When the separator routine receives control, register 1 contains the address of the parameter list, which contains the following:

Byte 0 This byte contains switches that indicate the type of output device, as follows:	
011.	2540 punch device
001.	1403, 3203-5, 3211, or 3800 printer device
010.	tape device with punch-destined output
000.	tape device with printer-destined output
Bytes 1-3 Reserved	
Bytes 4-7 This word is the address of the output DCB.	
Bytes 8-11 This word is the address of an 8-character field containing the jobname.	
Bytes 12-15 This word is the address of a 1-character field containing the output classname.	

Figure 15-3. Parameter List for Separator Routine

The parameter list points to a DCB; this DCB is established for the QSAM output data set, which is already open when the separator program receives control.

The address of the jobname and the address of the output classname are provided in the parameter list so they can be used in the separation records the separator routine writes.

Output from the Separator Routine

A separator routine can write any kind of separation identification. IBM supplies a routine that constructs block characters. (See “Using the Block Character Routine.”) The separator routine can punch as many separator cards, or print as many separator pages, as necessary.

The output from the separator program must conform to the attributes of the output data set. To find out what these attributes are, examine the open output DCB pointed to by the parameter list. The attributes are:

- Record format (fixed, variable, or undefined length)
- Record length
- Type of carriage control characters (machine, ANSI, or none)

For printer-destined output, a separator routine can begin its separator records on the same page as the previous job output, or skip to any subsequent page. However, the separator routine should skip at least one line before writing any records because in some cases the printer is still positioned on the line last printed.

After completing the output of the separation records, the separator routine should write sufficient blank records to force out the last separation record. This also allows the error exit routine to obtain control if an uncorrectable output error occurs while writing the last record. One blank record, for printer-destined output, and three blank records, for punch-destined output, are sufficient to force out the last record.

Writing an Output Separator Routine

If you write your own separator routine, it must conform to the following requirements:

- The routine must be named IEFSD094, and must reside in SYS1.LINKLIB or in a library concatenated to LINKLIB via a LNKLSTxx member of SYS1.PARMLIB.
- The routine must use standard entry and exit linkages, saving and restoring its caller's registers, and returning to its caller via the RETURN macro, with a return code in register 15.
- The routine must use the QSAM PUT macro in locate mode to write separation records to the output data set.
- The routine must use the GETMAIN and FREEMAIN macros to obtain and release the storage required for work areas.

- The routine must establish its own synchronous error exit routine, and place the exit address in the DCBSYNAD field of the output DCB. The error routine will receive control during output writing in case of an uncorrectable I/O error; it must set a return code of 8 (binary) in register 15 to indicate an unrecoverable output error.

If the separator routine completes processing successfully, it must set a return code of 0 in register 15, before returning to its caller.

Note: The separator routine receives control in problem-program state, but with a protection key of 0. Therefore, the routine must ensure data protection during its execution.

Using the Block Character Routine

For printer-destined output, the separator routine can use an IBM-supplied routine to construct separation records in a block character format. This routine is a reentrant module named IEFSD095 that resides in the module library SYS1.AOSB0.

The block character routine constructs block letters (A to Z), block numbers (0 to 9), and a blank. The separator routine furnishes the desired character string and the construction area. The block characters are constructed one line position at a time. Each complete character is contained in 12 lines and 12 columns; therefore, a block character area consists of 144 print positions. For each position, the routine provides either a space or the character itself.

The routine spaces 2 columns between each block character in the string. However, the routine does not enter blanks between or within the block characters. The separator routine must prepare the construction area with blanks or other desired background before entering the block character routine.

To invoke the IBM-supplied block character routine, the IBM-supplied separator routine executes the CALL macro with the entry point name of IEFSD095. Since the block characters are constructed one line position at a time, complete construction of a block character string requires 12 entries to the routine. Each time, the address of a 4-word parameter list is provided in register 1.

The parameter list contains the following:

Bytes 0-3	This fullword is the address of a field containing the desired character string in EBCDIC format.
Bytes 4-7	This fullword is the address of a field containing the line count as a binary integer from 1 to 12. This represents the line position to be constructed on this call.
Bytes 8-11	This word is the address of a construction area where the routine will build a line of the block character string. The required length in bytes of this construction area is $14n-2$, where n represents the number of characters in the string.
Bytes 12-15	This word is the address of a fullword field containing, in binary, the number of characters in the string.

The External Writer Cataloged Procedure

In order for an external writer to work in the MVS/JES2 or JES3 environment, it must be described to the system in a cataloged procedure residing in SYS1.PROCLIB; and it must be started by a START command, either from the system console or from within a problem program.

The IBM-supplied external writer is described and invoked by the cataloged procedure named XWTR, which can serve as the base or a model for a procedure you would write for your own output writer.

XWTR contains one step and consists of two JCL statements:

- The EXEC statement specifies the name of the external writer program to be executed.
- The DD statement defines the output data set.

Following is the actual XWTR procedure:

```
//IEFPROC EXEC PGM=IASXWROO,REGION=20K X
// PARM='PA,IEFSD094'
//IEFRDER DD UNIT=2400,VOLUME=(, , , 35), X
// DSNNAME=SYSOUT,DISP=(NEW,KEEP), X
// DCB=(BLKSIZE=133,LRECL=133,BUFL=133, X
// BUFNO=2,RECFM=FM)
```

The EXEC Statement

The generalized format for the EXEC statement is:

```
//IEFPROC EXEC PGM=IASXWROO[,REGION=nnnnnK,ADDRSPC=REAL]
// [,PARM='cxxxxxxxx[,seprname']]
```

The stepname must be IEFPROC, as shown. The parameter requirements are as follows:

PGM = IASXWROO

The name of the external writer load module. It must be IASXWROO, as shown.

REGION = nnnnnK

This parameter specifies the region size for the external writer program. The value nnnnn is a 1- to 5-digit number that is multiplied by 1K (1024 bytes) to designate the region size. The region size can vary according to the size of buffers and the size of your output writing routine. Insufficient region size will cause the external writer to abend.

ADDRSPC = REAL

Ensures that the external writer program will not be paged out during execution.

PARM = ['cxxxxxxxx[,seprname']]

This is a set of parameters for the output writing routine. The first part of the field can contain one to nine characters; the second part of the field contains the name of the output separator routine.

c

An alphabetic character, either **P** (for printer) or **C** (for card punch), that specifies the control characters for the class of output the output writing routine will process.

XXXXXXXX

From one to eight (no padding required) single-character SYSOUT classnames. These characters not only specify the classes the output writing routine will process, they also establish the priority for those classes, with the highest priority at the high-order end of the character string.

Note: If the START command includes classname parameters, they override all of the classnames coded here. If you do not code classnames on the procedure EXEC statement or the START command, then the external writer will wait for a MODIFY command from the operator before processing any output.

seprname

This is the name of the output separator routine to run with the output writing routine. IBM's output separator routine is named IEFSD094. If you write your own output separator routine, you must name it IEFSD094 and put its name in SYS1.LINKLIB (or a library concatenated to LINKLIB via a LNKLSTxx member of parmlib). In order for your separator routine to be invoked, you must code its name on this subparameter. If this subparameter is omitted, no output separator pages are produced.

The DD Statement

You must define the output data set the external writer will use in this statement. The generalized format for the DD statement is:

```

//IEFRDER DD UNIT=device,LABEL=(,type), X
// VOLUME=(,,volcount),DSNAME=aname, X
// DISP=(NEW,KEEP),DCB=(list of attributes), X
// UCS=(codeff,FOLD"ff,VERIFY"), X
// FCB=(image-id ,ALIGN )
// ,VERIFY

```

This must be the first DD statement in the procedure. The ddname should be IEFSD094, as shown. However, the system will always treat the first DD statement in this procedure as an output data set, regardless of the ddname. The parameter requirements are as follows:

UNIT = device

This specifies the printer, tape, card punch, or DASD device on which the output data set is to be written.

LABEL = type

This describes a data set label, if one is needed (for tape data sets only). If this parameter is omitted, a standard tape label is used.

VOLUME = (,,volcount)

Needed for tape data sets only, this parameter limits the number of tape volumes that this external writer can use during its entire operation.

DSNAME = anyname

This specifies a name for the output data set, so later steps in the procedure can refer to it. The data set name is required for the disposition of KEEP.

DISP = (NEW,KEEP)

The disposition of KEEP prevents deletion of the data set (tape and DASD only) at the end of the job step.

DCB = (list of attributes)

The DCB parameter specifies the characteristics of the output data set and the buffers. The BLKSIZE and LRECL subparameters are always required. The BUFL value, if you do not code it, is calculated from the BLKSIZE value. Other subparameter fields may be coded as needed; if they are not, the defaults are the QSAM default attributes. These are:

BUFNO—	Three buffers for the 2540 punch; two buffers for all other devices.
RECFM—	U-format, with no control characters.
TRTCH—	Odd parity, no data conversion, and no translation.
DEN—	Lowest density.
OPTCD—	Printer data checks are suppressed, and “select translate table” characters are printed as data. The IBM external writer does not support OPTCD = J, a 3800 printing subsystem specification.

UCS = (code[,FOLD][,VERIFY])

This specifies the code for a universal character set (UCS) image to be loaded into the UCS buffer.

FOLD causes bits 0 and 1 to be ignored when comparing characters between the UCS and print line buffers, thereby allowing lowercase alphabetic characters to be printed (in uppercase) by an uppercase print chain or train.

VERIFY causes the specified UCS image to be printed for verification by the operator.

The UCS parameter is optional, and is valid only when the output device is a 1403, a 3211, or a 3203-5 printer.

**FCB = (image-id ,ALIGN)
,VERIFY**

This causes the specified forms control buffer (FCB) image to be loaded into the FCB. ALIGN and VERIFY are optional subparameters that allow the operator to align forms. In addition, VERIFY causes the specified FCB image to be printed for visual verification. The FCB parameter is valid only for a 3203-5, 3211, or 3800 printer; otherwise, it is ignored.

See *MVS/XA JCL* for more information on the parameters mentioned here.

Special Printer Output Considerations: To process output jobs that require special chains for printing, you should have specific classes for each different print chain. You can specify the desired chain in your output writer procedure, and when that output writer is started, the chain will be loaded automatically. (Printers used with special chains should be named with esoteric group names as defined at sysgen time. See “The Eligible Device Table.”)

Following is an example of the JCL needed to define a special print chain in a cataloged procedure for an external writer.

```
//IEFPROC EXEC PGM=IASXWROO,REGION=20K,PARM='PDEG,IEFSD094'  
//IEFRDER DD UNIT=SYSPR,DSNAME=SYSOUT,FCB=(STD2,ALIGN), X  
// UCS=P11,DISP=(,KEEP), X  
// DCB=(BLKSIZE=133,LRECL=133,BUFL=133, X  
// BUFNO=2,RECFM=FM)
```

In this example, the UCS DD parameter requests the P11 print chain for data sets in the SYSOUT classes D, E, and G.

If the output device is a 3211 or a 3203-5, a UCS or FCB image can be loaded dynamically between the printing of data sets. Therefore, you can specify a mixture of data sets using different images in a single output class for this device. This will probably require mounting trains and changing forms, however, so it might not be desirable.

When the output device is a 1403 or 3800, the UCS image or 3800 attributes are specified at START XWTR time; they cannot be changed until the writer is stopped. Therefore, all data sets within an output class must be printed using the same train.

The FCB image is ignored when the 1403 printer is the output device.

External writer output to an IBM 3800 Printing Subsystem can also make use of the CHARS, COPIES, FLASH, and MODIFY JCL parameters on the DD statement. For information about using these parameters, see *IBM 3800 Printing Subsystem Programmer's Guide*. The coding rules and defaults are documented in *MVS/XA JCL Reference*.



Chapter 16. The Virtual Fetch Service

Virtual fetch is a service that reduces the time required to locate a load module and bring it into storage for execution. The virtual fetch service runs as a started task in its own address space. When virtual fetch manages a module, it makes a reformatted copy of the module and moves it to a VIO data set that it owns.

Moving load modules from a data base device complex to paging devices can reduce contention for data base channels in your installation. At the same time, the VIO data set is quickly accessible when needed, and virtual fetch relocates the load module's address constants efficiently.

Thus, you can use virtual fetch to improve the responsiveness of interactive subsystems or other large-scale processing programs. Using the virtual fetch service on a system with extended storage will usually result in greater responsiveness for those load modules in the virtual fetch data set that are frequently referenced.

Virtual fetch could handle any load module that is executable and not in overlay format. However, a load module that virtual fetch is to manage must also be read-only. Because virtual fetch runs as a started task, you must define these modules to the service by placing DD statements in the JCL procedure that starts the virtual fetch address space. The DD statements identify the load libraries virtual fetch is to manage.

You will need to place the virtual fetch JCL procedure in SYS1.PROCLIB. You must define this procedure as described in the section "Installation Support." You can further control access to the virtual fetch service by placing the virtual fetch initialization program in a RACF- or password-protected data set.

Functions of Virtual Fetch

Once the service has been initialized, any user can obtain modules from virtual fetch, regardless of the caller's state or key.

The virtual fetch functions — initialization, build, find, and get — are described in the following pages.

The Virtual Fetch Initialization Function: CSVVFCRE

The initialization function establishes the virtual fetch service address space. Initialization consists of starting program CSVVFCRE any time after JES initialization.

CSVVFCRE initializes the cross-memory environment by creating the virtual fetch cross-memory entry table. The entry table allows callers in other address spaces to issue program call (PC) instructions to the virtual fetch search routine (CSVVFSCH). CSVVFSCH runs in the virtual fetch service address space.

Another cross-memory entry table, created by PC/AUTH during MVS system initialization, allows virtual fetch interface routines to issue PC instructions to get into supervisor state.

CSVVFCRE also creates the virtual fetch control block (VFCB) in the common area, creates a read-only VIO data set of reformatted load modules, and creates a hash table of virtual fetch directory entries (VFDEs). Each VFDE points to one of the reformatted load modules in the VIO data set.

The hash table resides in the virtual fetch service address space. Virtual fetch owns the VIO data set and can read it from any address space.

Program CSVVFCRE is also active during a request to refresh virtual fetch modules; see "Refreshing Virtual Fetch."

If the initialization of virtual fetch is unsuccessful, CSVVFCRE returns to the caller with a return code in register 15. The codes and their meanings are:

Code	Meaning
4	Another virtual fetch service address space already exists. CSVVFCRE ignores this initialization request and issues message CSV108I.
8	Either the caller requested virtual fetch service address space initialization without providing load modules, or the caller did not provide any valid directory entries for the load modules. CSVVFCRE ignores this initialization request and issues message CSV103I.
12	CSVVFCRE received a processing error from ASM's ASSIGN-LGN service when allocating the VIO data set. CSVVFCRE ignores this initialization request and issues message CSV117I.
16	CSVVFCRE received a processing error from RSM's ASSIGN-NUL service when creating a VIO window. CSVVFCRE ignores this initialization request and issues message CSV117I.
20	CSVVFCRE received a processing error from RSM's MOVEOUT-disconnect service when writing a reformatted load module to the VIO data set. CSVVFCRE ignores this initialization request and issues message CSV117I.
24	CSVVFCRE was not invoked as a started task.

Once the virtual fetch service has been initialized, a problem program can call virtual fetch to perform three functions:

- To manage a module for the caller (a BUILD request)
- To determine if virtual fetch is managing a module for the caller (a FIND request)
- To pass control to a module (a GET request)

The environment prior to the initialization of virtual fetch is illustrated in Figure 16-1. Figure 16-2 illustrates the environment after virtual fetch initialization.

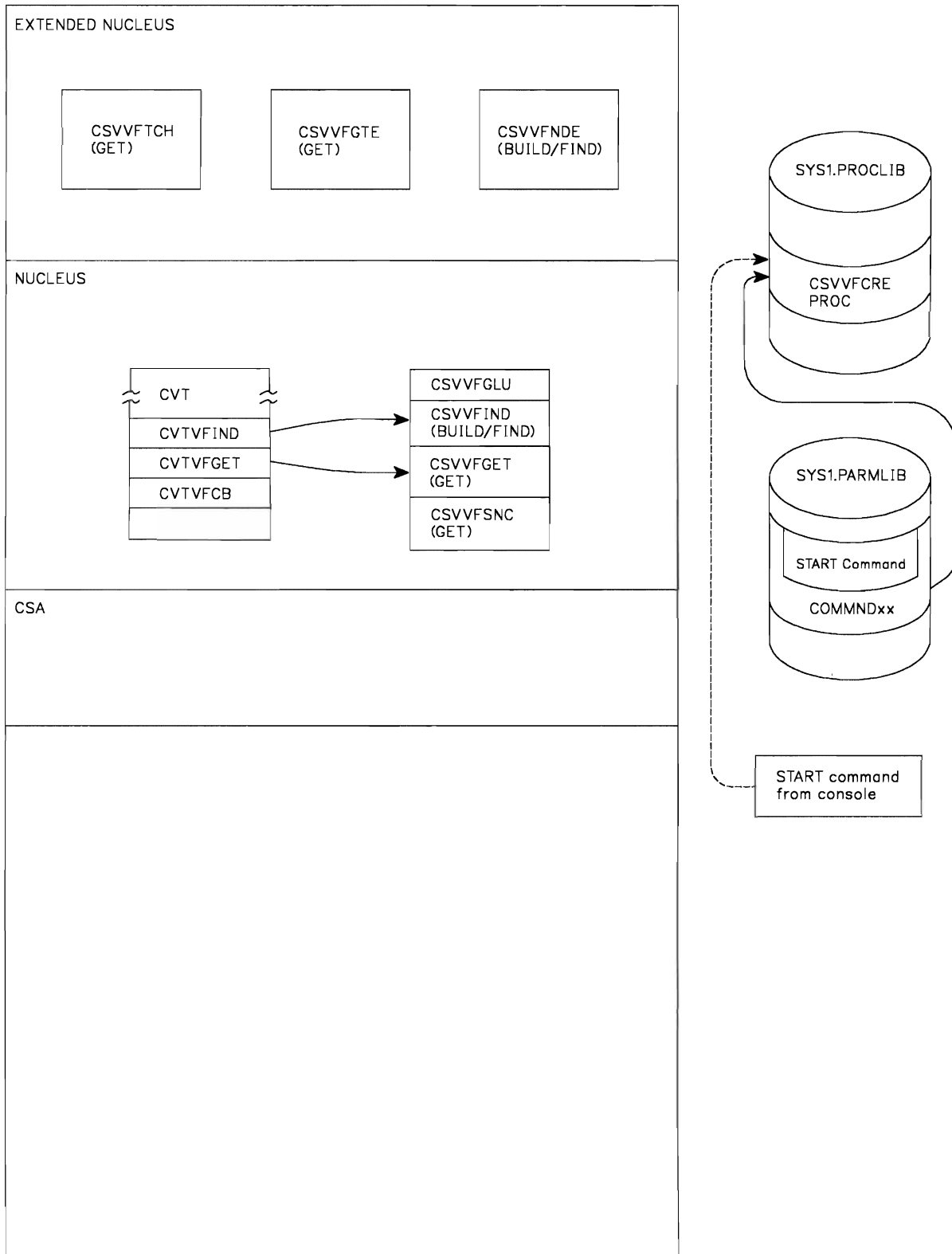


Figure 16-1. Environment Prior to Virtual Fetch Initialization

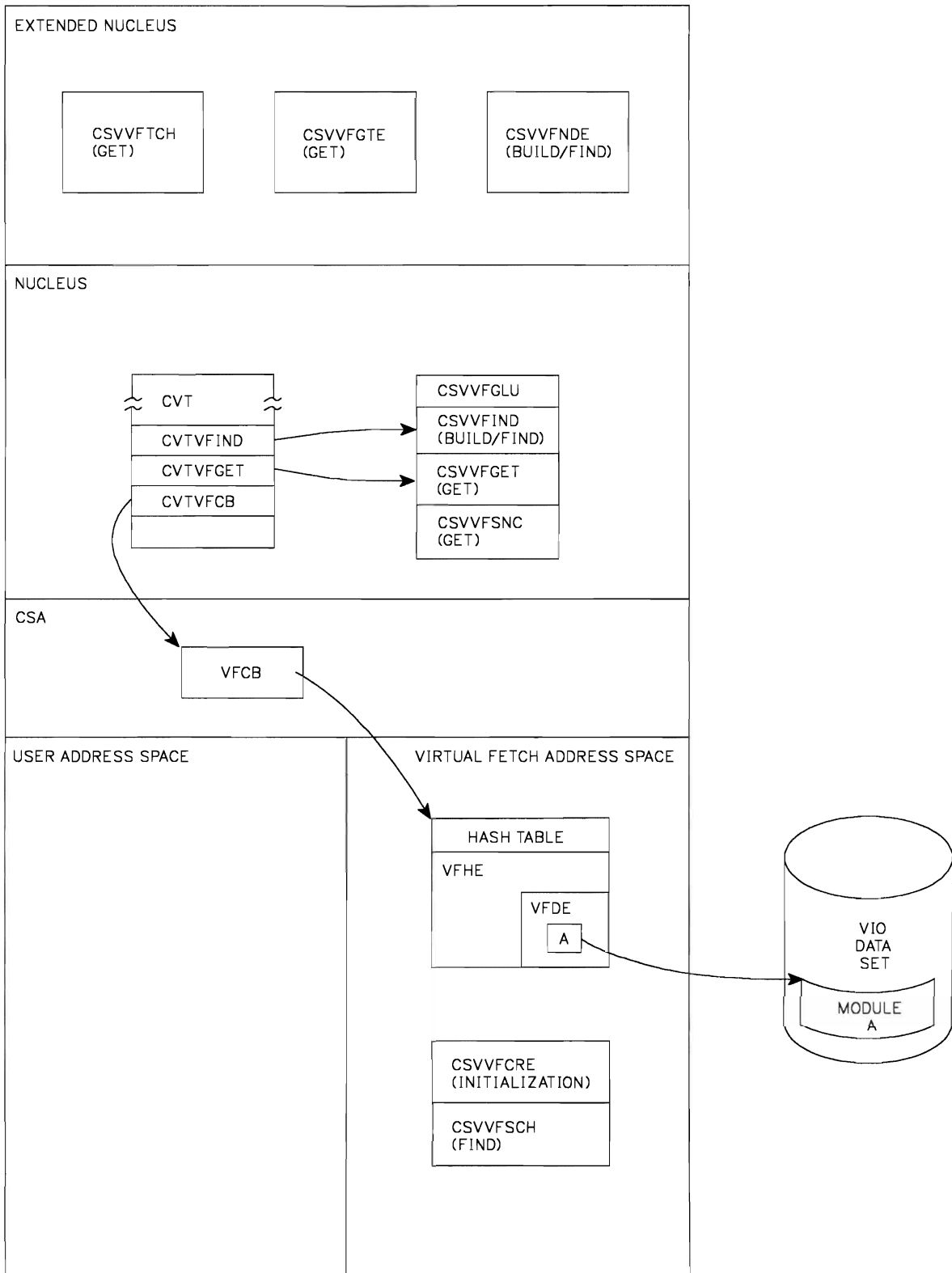


Figure 16-2. Environment After Virtual Fetch Initialization

The Virtual Fetch Build Function: CSVVFNDE

The virtual fetch build function (module CSVVFNDE) creates the virtual fetch work area (VFWK) for the named module, in the caller's address space. When called for the first time, it also creates the virtual fetch vector table (VFVT) for the caller.

The VFWK identifies the module that the caller wishes to access via the virtual fetch find and get functions.

The Virtual Fetch Find Function: CSVVFSCH

FIND requests invoke module CSVVFNDE to perform the find function. The find function locates (via module CSVVFSCH) the virtual fetch directory entry (VFDE) for the named module in the virtual fetch service address space and copies the VFDE into the VFWK in the caller's address space. A return code indicates whether or not a VFDE for the named module was found.

The Virtual Fetch Get Function: CSVVFGET, CSVVFTCH

The virtual fetch get function uses the VFDE in the VFWK to obtain a copy of the named module from virtual fetch's read-only VIO data set. Virtual fetch then completes relocation of address constants and passes control to the named module. Return flags in the virtual fetch parameter list (VFPM field VFPMRTN) indicate if the module was invoked.

The get function is activated via a call to a virtual fetch routine (CSVVFGET) pointed to by field CVTVFGET in the CVT. CSVVFGET invokes module CSVVFGTE.

Using the VFDE obtained by CSVVFNDE, CSVVFGTE calls CSVVFTCH to page in a copy of the named module and complete relocation of its address constants.

CSVVFGTE then passes control to the named module. When the named module completes, CSVVFGTE cleans up and returns to the caller of CSVVFGET.

Installation Support for the Virtual Fetch Service

If the virtual fetch service address space terminates, it can be restarted by a START command from the operator's console.

Termination of the virtual fetch service address space does not affect the caller's address space; callers are informed that modules cannot be obtained from virtual fetch. In the case of a FIND request, CSVVFNDE returns a code of 8 in register 15. In the case of a GET request, CSVVFGTE sets on (1) bit VFPMRESH in VFPMRTN. Callers should then use conventional linkage mechanisms to invoke their modules.

When load modules are managed by virtual fetch, they can be updated. That is, if you make changes to the partitioned data sets that are input to virtual fetch, virtual fetch can create a new VIO data set and hash table to reflect these changes.

The virtual fetch refresh process makes all updates take effect simultaneously. The refresh process is not, however, automatic. When you want to update a module managed by virtual fetch, you must explicitly request virtual fetch refresh processing.

Virtual fetch cannot serialize refresh requests or initialization processing with any updates to the load modules it is managing. You must coordinate the updating of load modules with requests for virtual fetch processing.

Starting Virtual Fetch

The program that creates the virtual fetch service address space is CSVVFCRE. The operator can start CSVVFCRE by issuing a START command, or you can place the START command in a COMMNDxx member of SYS1.PARMLIB; in this case, the operator starts CSVVFCRE when he includes CMD=xx in his response to the "SPECIFY SYSTEM PARAMETERS" message during an IPL. The JCL procedure that executes CSVVFCRE must be in SYS1.PROCLIB. In either case, virtual fetch must be invoked as a started task.

Only one virtual fetch service address space can exist at any one time. Therefore, include in the virtual fetch service ALL the modules needed by any users of virtual fetch.

Virtual fetch's build, find and get processing operate at the priority of the calling program.

The procedure that executes CSVVFCRE must contain DD statements with names in the range VFIN00 to VFIN99. These DD statements define partitioned data sets containing the load modules to be managed by the virtual fetch service.

Although JCL DD concatenation is not supported, the DD statements are opened in the order implied by their ddnames (for example, VFIN04 before VFIN07), giving the effect of up to 100 concatenated data sets.

Virtual fetch attempts to access, in numerical order, every DD name from VFIN00 to VFIN99. Therefore, the order in which the DD statements appear is irrelevant.

The following example of a started procedure for virtual fetch illustrates the use of the VFINnn names:

```
//IEFPROC   EXEC   PGM=CSVVFCRE,TIME=1440
//VFIN00    DD     DSN=LIB1,DISP=SHR
//VFIN03    DD     DSN=LIB2,DISP=SHR
//VFIN02    DD     DSN=LIB4,DISP=SHR
//SYSUDUMP  DD     SYSOUT=A
/*
```

In this example, the sequence of the data sets as opened by virtual fetch is LIB1, LIB4, and LIB2.

Notes:

- 1. You should not code `FREE=CLOSE` on `DD` statements describing the virtual fetch libraries. The resulting deallocation of the libraries makes them unavailable for subsequent virtual fetch processing.*
- 2. If there are identically-named modules in the virtual fetch libraries, virtual fetch recognizes only the first occurrence of the module. All subsequent occurrences are ignored.*
- 3. If the data sets containing the modules virtual fetch manages are in private catalogs, those catalogs are allocated for the life of the virtual fetch service address space.*

Refreshing Virtual Fetch

You can rebuild virtual fetch's VIO data set and hash table to reflect changes made to the load modules in your source libraries. The refresh process makes all updates available to virtual fetch users at the same time.

After CSVVFCRE builds the VIO data set and the hash table, it can refresh them. On receiving a refresh request via program CSVVFRSH, CSVVFCRE builds a new VIO data set and hash table for the modules it obtains from the input data sets. Virtual fetch then activates the new VIO data set and hash table, frees the old VIO data set and hash table, issues a message to the operator that the refresh process is complete, and waits for another request for refresh processing. The wait is terminated either by the operator cancelling CSVVFCRE or by CSVVFCRE receiving another refresh request.

If CSVVFCRE cannot complete the refresh, the existing VIO data set and hash table remain active, if possible. If the existing VIO data set cannot remain active, CSVVFCRE cleans up and abends.

The POST that initiates the refresh process is issued by program CSVVFRSH. To invoke the refresh function of virtual fetch, you submit a job to execute program CSVVFRSH (`// EXEC PGM=CSVVFRSH`).

During refresh processing, virtual fetch uses only those libraries specified at the time of its initialization. Because CSVVFRSH uses authorized facilities (cross-memory POST, for example), the program must reside in an APF-authorized library, and you must linkedit CSVVFRSH using a nonzero authorization code.

It is your responsibility to run CSVVFRSH when you determine that a refresh is needed. If you want the operator to control the refresh process, you can put CSVVFRSH in a password- or RACF-protected data set, and start it with a JCL procedure in SYS1.PROCLIB.

Considerations When Using Virtual Fetch

- To use the virtual fetch service, the calling program must:
 - Have only a single jobstep TCB in its address space
 - Be in TCB mode
 - Not be in cross-memory mode
 - Not have any outstanding FRRs
 - Not hold any locks

The virtual fetch routines that CVTVFIND and CVTVFGET point to cannot be invoked by an ESTAE-type recovery routine or by an FRR.

- During virtual fetch initialization or refresh processing, do not change the virtual fetch input libraries; the resulting load modules that virtual fetch places in its VIO data set could be invalid.
- Modules managed by virtual fetch cannot use the checkpoint/restart facility. In addition, job steps using virtual fetch services cannot use the checkpoint/restart facility.
- IDENTIFY cannot be used against modules that have been obtained via a virtual fetch GET request.
- Modules that receive control by means of a GET request are not reused: a fresh copy is obtained for each GET request; only one virtual fetch copy can exist in an address space, and only one caller at a time can use the virtual fetch copy.
- When the build, find, and get functions are invoked, register 1 points to a parameter list (VFPM) as mapped by macro IHAVFPM. The format of the parameter list is shown in Figure 16-3.

Length	Name	Description																								
72	VFPMSAVE	18-word register save area that the caller wants passed to the requested program (module).																								
4	VFPMREG1	A value that the caller wants virtual fetch to pass to the requested program in register 1.																								
8	VFPMNAME	Name of the requested program, left-justified with trailing blanks.																								
1	VFPMLVL	Release level of parameter list. The caller must set this field to the current parameter list level (presently, 0).																								
1	VFPMFUNC	Function virtual fetch is to perform. Decimal 1 = BUILD function (VFPMBLD) Decimal 2 = FIND function (VFPMFIND) Decimal 3 = GET function (VFPMGET)																								
1	VFPMFLAG	Flag byte.																								
		<table border="1"> <thead> <tr> <th>Bit</th> <th>Name</th> <th>Meaning When Set</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>VFPMGETM</td> <td></td> </tr> <tr> <td></td> <td>On -</td> <td>Fresh module storage is GETMAINED and FREEMAINED on each invocation.</td> </tr> <tr> <td></td> <td>Off -</td> <td>A GETMAIN is performed for module storage on the first invocation of the module only. Virtual fetch performs additional GETMAINS only when a refreshed module requires additional storage. Storage is released by means of a page release.</td> </tr> <tr> <td>1-7</td> <td></td> <td>Reserved.</td> </tr> </tbody> </table>	Bit	Name	Meaning When Set	0	VFPMGETM			On -	Fresh module storage is GETMAINED and FREEMAINED on each invocation.		Off -	A GETMAIN is performed for module storage on the first invocation of the module only. Virtual fetch performs additional GETMAINS only when a refreshed module requires additional storage. Storage is released by means of a page release.	1-7		Reserved.									
Bit	Name	Meaning When Set																								
0	VFPMGETM																									
	On -	Fresh module storage is GETMAINED and FREEMAINED on each invocation.																								
	Off -	A GETMAIN is performed for module storage on the first invocation of the module only. Virtual fetch performs additional GETMAINS only when a refreshed module requires additional storage. Storage is released by means of a page release.																								
1-7		Reserved.																								
1	VFPMRTN	Return indicator flag byte set by the GET function. If VFPMRTN is zero, the module specified in VFPMNAME has executed and register 15 contains its return code. If VFPMRTN is not zero, a single bit will be set, as follows:																								
		<table border="1"> <thead> <tr> <th>Bit</th> <th>Name</th> <th>Meaning When Set</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>VFPMBUSY</td> <td>The module is in use. The module does not receive control. The caller should retry by invoking the find function.</td> </tr> <tr> <td>1</td> <td>VFPMRESH</td> <td>GET is not able to obtain the requested module. The module does not receive control. The caller should retry by invoking the find function.</td> </tr> <tr> <td>2</td> <td>VFPMAPF</td> <td>An authorized user tried to invoke a module that originally came from a non-APF-authorized library. The module does not receive control. The caller should obtain the module from an authorized library.</td> </tr> <tr> <td>3</td> <td>VFPMBADP</td> <td>Virtual fetch detected an invalid parameter list. Virtual fetch attempts no further processing. The caller should examine the VFPM for incorrect values and retry by invoking the get function.</td> </tr> <tr> <td>4</td> <td>VFPMBADE</td> <td>Virtual fetch encountered an environmental error. (GETMAIN failed, ESTAE failed, etc.). The module does not receive control. Perform clean-up and retry by invoking the find function.</td> </tr> <tr> <td>5</td> <td>VFPMAPPL</td> <td>The requested program abnormally terminated. This bit should be examined whenever the caller's recovery routine gains control. Retry by invoking the find function.</td> </tr> <tr> <td>6-7</td> <td></td> <td>Reserved.</td> </tr> </tbody> </table>	Bit	Name	Meaning When Set	0	VFPMBUSY	The module is in use. The module does not receive control. The caller should retry by invoking the find function.	1	VFPMRESH	GET is not able to obtain the requested module. The module does not receive control. The caller should retry by invoking the find function.	2	VFPMAPF	An authorized user tried to invoke a module that originally came from a non-APF-authorized library. The module does not receive control. The caller should obtain the module from an authorized library.	3	VFPMBADP	Virtual fetch detected an invalid parameter list. Virtual fetch attempts no further processing. The caller should examine the VFPM for incorrect values and retry by invoking the get function.	4	VFPMBADE	Virtual fetch encountered an environmental error. (GETMAIN failed, ESTAE failed, etc.). The module does not receive control. Perform clean-up and retry by invoking the find function.	5	VFPMAPPL	The requested program abnormally terminated. This bit should be examined whenever the caller's recovery routine gains control. Retry by invoking the find function.	6-7		Reserved.
Bit	Name	Meaning When Set																								
0	VFPMBUSY	The module is in use. The module does not receive control. The caller should retry by invoking the find function.																								
1	VFPMRESH	GET is not able to obtain the requested module. The module does not receive control. The caller should retry by invoking the find function.																								
2	VFPMAPF	An authorized user tried to invoke a module that originally came from a non-APF-authorized library. The module does not receive control. The caller should obtain the module from an authorized library.																								
3	VFPMBADP	Virtual fetch detected an invalid parameter list. Virtual fetch attempts no further processing. The caller should examine the VFPM for incorrect values and retry by invoking the get function.																								
4	VFPMBADE	Virtual fetch encountered an environmental error. (GETMAIN failed, ESTAE failed, etc.). The module does not receive control. Perform clean-up and retry by invoking the find function.																								
5	VFPMAPPL	The requested program abnormally terminated. This bit should be examined whenever the caller's recovery routine gains control. Retry by invoking the find function.																								
6-7		Reserved.																								

Figure 16-3. Virtual Fetch Parameter List

Notes:

1. *The caller must set to zero all reserved bits in fields VFPMFLAG and VFPMRTN. Virtual fetch uses the flag bits in VFPMRTN to indicate why a module did not receive control, or that it did receive control but abnormally terminated. Only one flag bit is on at a time.*
2. *The VFPMFUNC field of the parameter list communicates to virtual fetch the function (build, find, or get) that virtual fetch is to perform. You can request only one function per invocation of virtual fetch. After issuing BUILD requests for the required modules, a program must invoke virtual fetch twice more (find and get functions) each time it uses any of the modules.*

Programming Conventions for Using Virtual Fetch

- A caller of virtual fetch must provide a standard 18-word save area, pointed to by register 13; virtual fetch will save the caller's registers in this area. The virtual fetch parameter list contains another 18-word save area, VFPMSAVE, where the called module can save its caller's registers on entry.
- The called module receives control via standard entry linkage. That is, register 1 contains the value from field VFPMREG1 of the virtual fetch parameter list, register 13 points to a save area (VFPMSAVE), register 14 contains the return address, and register 15 contains the entry point address. Any unused high-order address bits of registers 1, 13 and 14 must be zero.
- The find and get functions may be invoked in any key, state, or addressing mode; however, virtual fetch assumes that the PSW key matches the caller's TCB key and the jobstep TCB key.

The environment that exists after an initial BUILD request and a subsequent FIND request is illustrated in Figure 16-4.

Requesting Dumps When Using Virtual Fetch

To get a dump of the in-use modules managed by virtual fetch, specify the JPA (job pack area) option either on the SNAP macro or in the appropriate parmlib member.

Note: You may request a certain module from virtual fetch and, concurrently, invoke the same module via a conventional contents supervision mechanism (LINK, for example). If you then request a dump for that module, both copies will be dumped.

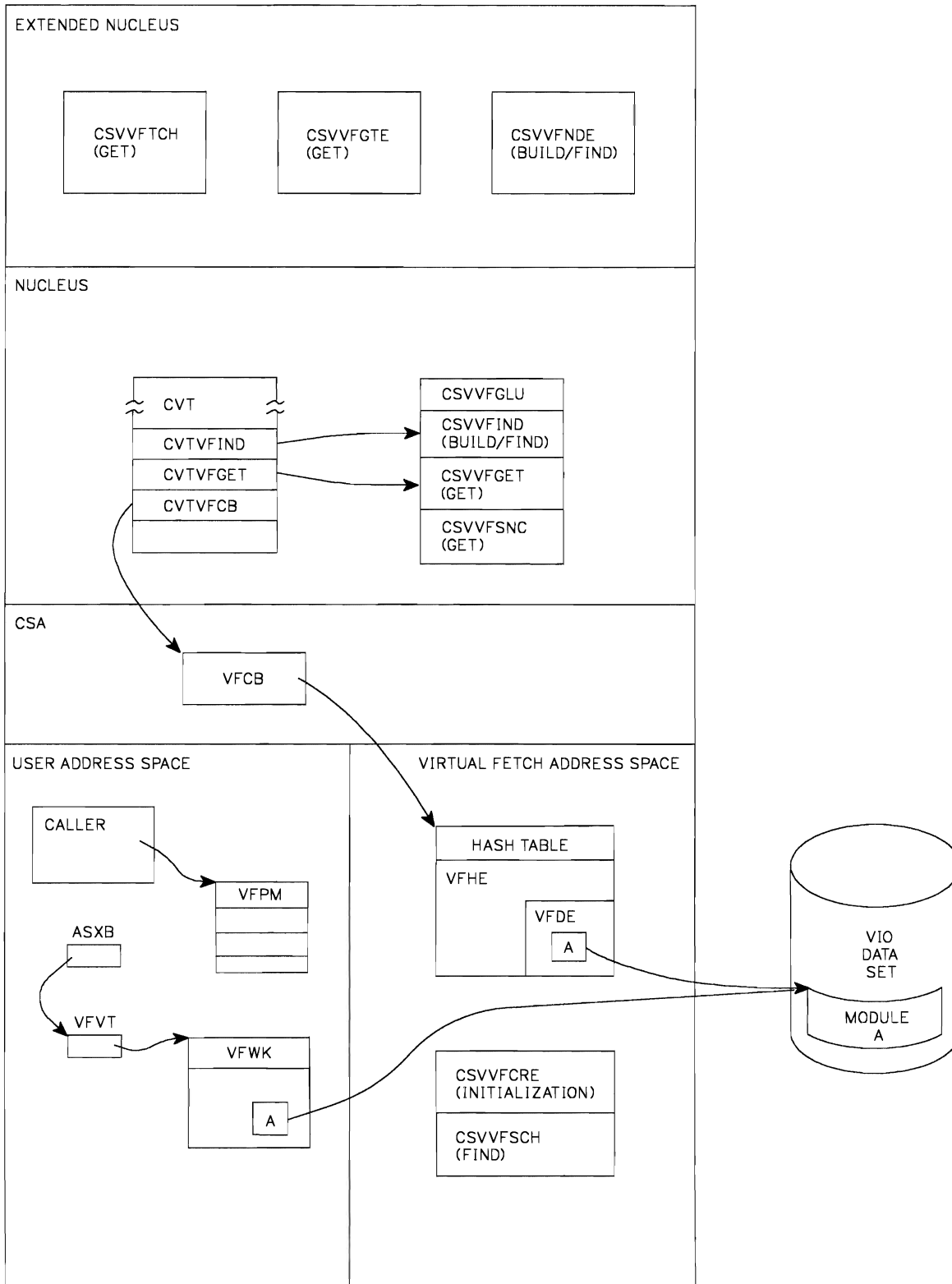


Figure 16-4. Environment After a BUILD/FIND Request

BUILD Request for Virtual Fetch

When a program calls virtual fetch to request that it manage a module (a BUILD request):

- Register 15 is loaded from CVTVFIND prior to invoking virtual fetch.
- VFPMFUNC is set to VFPMBLD (decimal 1).
- Register 14 is the return register. Any unused high-order address bits of register 14 must be zero.
- Register 1 contains the address of the VFPM. Any unused high-order address bits of register 1 must be zero.
- Register 13 contains the address of a standard 18-word save area. Any unused high-order address bits of register 13 must be zero.
- VFPMNAME is set to the name of the program virtual fetch is to manage, left-justified with trailing blanks.
- VFPMMLVL is set to the current parameter list level (presently, 0).

When virtual fetch returns control, register 15 contains a return code. The return codes and their meanings are:

Code	Meaning
0	Virtual fetch is able to manage the module.
8	Virtual fetch's local control blocks (VFWKs or VFVT) are unusable.
16	Virtual fetch detected an input parameter error.
20	Virtual fetch detected an environmental error.

Note: A subsystem could invoke virtual fetch's build function during its own initialization processing. The build function does not depend on the prior initialization of the virtual fetch service address space.

FIND Request for Virtual Fetch

When a program calls virtual fetch to find the VFDE for a module (a FIND request):

- Register 15 is loaded from CVTVFIND prior to invoking virtual fetch.
- VFPMFUNC is set to VFPMFIND (decimal 2).
- Register 14 is the return register. Any unused high-order address bits of register 14 must be zero.
- Register 1 contains the address of the VFPM. Any unused high-order address bits of register 1 must be zero.
- Register 13 contains the address of a standard 18-word save area. Any unused high-order address bits of register 13 must be zero.

- VFPNAME is set to the name of the program whose VFDE virtual fetch is to find. The name is left-justified with trailing blanks.
- VFPMLVL is set to the current parameter list level (presently, 0).

When virtual fetch returns control, register 15 contains a return code. The return codes and their meanings are:

Code	Meaning
0	Virtual fetch found the VFDE.
4	Either the module is not managed by virtual fetch or the virtual fetch service's copy of the module is not usable.
8	The local virtual fetch control blocks (VFWKs or VFVT) contain errors, or the virtual fetch service address space is not operational.
12	The requested VFDE is not in the current virtual fetch hash table.
16	Virtual fetch detected an input parameter error.
20	Virtual fetch encountered an environmental error.

Requesting the virtual fetch find function causes virtual fetch to locate and validate the local virtual fetch work area (VFWK) for the requested module. If necessary, virtual fetch copies the VFDE from the current hash table (in the virtual fetch service address space) into the VFWK.

GET Request for Virtual Fetch

When a program calls virtual fetch to pass control to a program (a GET request):

- Register 15 is loaded from CVTVFGET prior to invoking virtual fetch.
- VFPFUNC is set to VFPGET (decimal 3).
- VFPGETM may be on or off.
- VFPMSAVE is available for use by the called program as a standard save area.
- VFPREG1 contains data to be passed to the called program in register 1.
- Register 14 is the return register. Any unused high-order address bits of register 14 must be zero.
- Register 1 contains the address of the VFP. Any unused high-order address bits of register 1 must be zero.
- Register 13 contains the address of a standard 18-word save area. Any unused high-order address bits of register 13 must be zero.
- VFPNAME is set to the name of the program virtual fetch is to invoke. The name is left-justified with trailing blanks.

- VFPMLVL is set to the current parameter list level (presently, 0).
- VFPMRTN is set to zero.

When virtual fetch returns control to its caller, the caller should examine field VFPMRTN in the VFPM. If field VFPMRTN is zero, then the module specified in VFPMNAME received control. Register 15 contains the return code from the specified module. If the module received control, it did so in the state and key, and with the key mask, of the caller of virtual fetch.

If VFPMRTN is not zero, either the named module did not receive control or it abnormally terminated while executing. Examine field VFPMRTN to determine the nature of the error.

Note: Modules managed by virtual fetch within the caller's address space that are in use when a GET request is made are considered busy (VFPMBUSY is ON) and are not given control. This applies to all modules, regardless of their attributes.

Users of virtual fetch may turn off bit VFPMGETM in the parameter list to keep virtual fetch from issuing GETMAINs and FREEMAINs for module storage. (See VFPMFLAG in Figure 16-3.) In order to benefit from the performance gain thus provided, the caller of virtual fetch must be able to tolerate the fact that virtual storage in the caller's address space, for the modules virtual fetch manages, is not freed by means of the FREEMAIN macro.

The environment that exists after a GET request is illustrated in Figure 16-5.

Figure 16-6 is a general illustration of the use of virtual fetch.

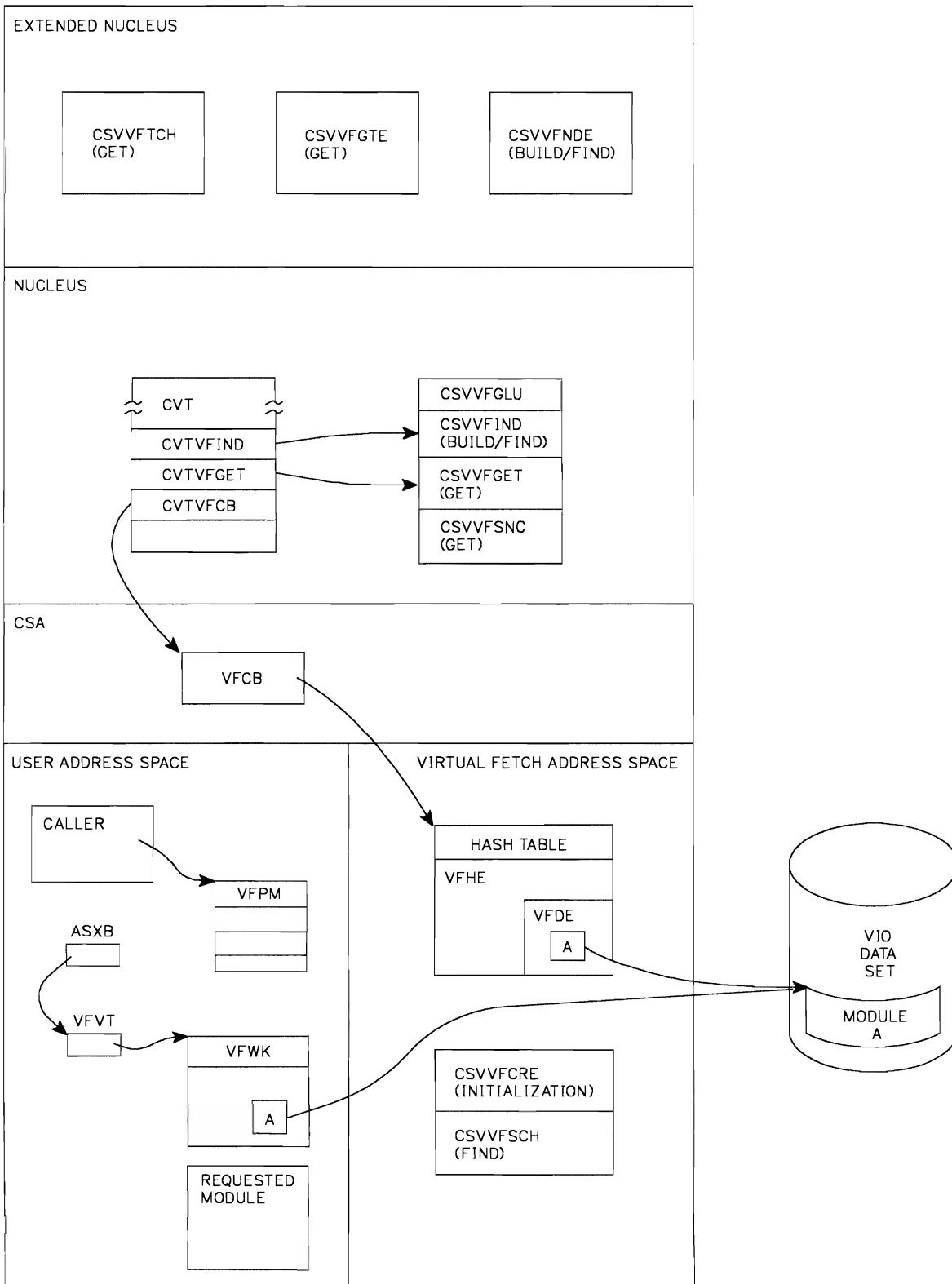


Figure 16-5. Environment After a GET Request

Let VFPMPTR be the name of register 1, which points to the start of the VFPM area.

```

      .
      .
      .
XC   VFPM(VFPMLEN,VFPMPTR),VFPM(VFPMPTR)  *Clear the VFPM area

MVI  VFPMFUNC(VFPMPTR),VFPMBLD           *Indicate BUILD request.
                                         (For a FIND request use
                                         VFPMFIND. For a GET
                                         request use VFPMGET.)

MVC  VFPMNAME(8,VFPMPTR),MODNAME         *Move the name of the
                                         desired module into
                                         the VFPM.

L    R15,CVTVFIND                        *Set up entry point
                                         address. (For a GET
                                         request use CVTVFGET.)

BALR R14,R15                             *Invoke virtual fetch
                                         function.

LTR  R15,R15                             *Examine return code.
                                         (For a GET request you
                                         first examine the
                                         VFPMRTN field of the
                                         VFPM.)

BZ   OK
      .
      . (Process any errors.)
      .
OK   (Normal processing continues)
      .
      .
      .

MODNAME DC CL8                            *Name of requested module
VFPMPTR EQU 1
```

Figure 16-6. A Program Using Virtual Fetch



Chapter 17. MVS Dumping Services

Along with codes and messages, dumps are among the most important problem-solving tools available to a system programmer.

Using the MVS/XA operating system, a data processing installation can do a great deal of work. The potentially large workload can lead to a correspondingly large number of dumps. In the MVS/XA environment, the extended architecture has the potential of expanding to 2 gigabytes – more than 2 billion bytes – for each user's address space. Virtual storage can be very large indeed. Uncontrolled dumping of storage can become a major cost factor in system time spent in taking dumps, in hours spent reading dumps, and in the cost of printing the dumps.

MVS dumping services provide dumps for debugging problems within the operating system and application programs. The dumping services help you control the number and size of dumps taken in your installation, and so reduce the costs of taking and debugging dumps.

To use the dumping services to your best advantage, you may want to tailor the functions to fit your installation. With MVS dumping services and associated functions, you can:

- Define data sets to be used as dump data sets at IPL, or in JCL, or via the DUMPDS command
- Suppress SVC dump and SYSMDUMP dumps using dump analysis and elimination (DAE)
- Suppress dumps automatically according to abend code, via SLIP
- Tailor or suppress types of dumps using the dump options and operator commands
- Tailor or suppress individual ABEND dumps in pre-dump exits
- Control individual SVC and SYSMDUMP dumps in post-dump exits

This chapter describes these tasks and presents some recommendations on how best to approach them.

To help you decide how and where best to modify the dumping services, you need to know how they work. Some general information about the kinds of dumps available in the MVS and MVS/XA environment follows.

MVS Dumps

In the MVS operating system, there are four kinds of dumps:

- SNAP** produced by SNAP processing in response to the SNAP macro coded in a problem program.
- MVS dumping services do not apply directly to SNAP dumps. A SNAP dump is not normally produced as a result of an error condition. *MVS/XA Supervisor Services and Macro Instructions* contains information about the SNAP macro and dump. *MVS/XA Debugging Handbook* includes examples of SNAP dumps.
- STAND-ALONE** produced by the AMDSADMP service aid when the operator IPLs a SADMP residence volume. Used as a last resort in the event of a major system error.
- MVS dumping services do not apply directly to stand-alone dumps; a stand-alone dump is not produced by MVS. *MVS/XA Service Aids* describes how to request stand-alone dumps and how to use AMDPRDMP to format and print them. *MVS/XA Diagnostic Techniques* contains information on debugging stand-alone dumps.
- ABEND** produced by SNAP/ABDUMP processing during abnormal termination of a job or job step.
- SVC** produced by SDUMP processing when a system routine has an error that should be documented for problem determination.

Note: JES3 also has facilities for SNAP and ABEND dumping: RJPSNPS, CBPRNT and DEBUG parameters format and write snap dumps of channel-end data and JES3 control blocks to specific data sets; and the JESSADMP DD statement defines a data set to receive an ABEND dump. See *MVS/XA JES3 Diagnosis* for details.

The MVS dumping services deal with **ABEND** and **SVC** dumps.

ABEND Dumps

ABEND dumps, also called **user dumps**, may help you to identify a problem in an application program. An ABEND dump is associated with task-mode error detection and recovery. To request an **ABEND dump** you can use any of three macros: ABEND, CALLRTM, or SETRP. There are three types of ABEND dumps:

SYSABEND	user-oriented, formatted dumps of task-related data. Produced by ABDUMP code.
SYSUDUMP	user-oriented, formatted dumps; usually more limited than SYSABEND dumps. Produced by ABDUMP code.
SYSMDUMP	user-oriented, unformatted (machine-readable) dumps. Produced by SDUMP code at the request of ABDUMP.

Defining ABEND Dump Data Sets

You define the data sets for ABEND dumps in the JCL for each job step. The ddname corresponds to the type of dump ABDUMP processing is to produce in case of an abend. SYSABEND and SYSUDUMP dumps are formatted by ABDUMP routines and are usually processed as system output (SYSOUT) by the job entry subsystem. SYSMDUMPs are stored on dump data sets until processed by the print dump service aid (AMDPRDMP) or by Interactive Problem Control System (IPCS).

Requesting ABEND Dumps

Requesting SYSABEND AND SYSUDUMP Dumps: To generate the SYSABEND and SYSUDUMP dumps, the JCL for a job step must include a corresponding DD statement for an output data set. For example:

```
//STEPNM EXEC PGM=ABC  
//SYSUDUMP DD SYSOUT=A
```

In the above example, if the job step, STEPNM, has an abnormal termination, the system takes a SYSUDUMP. The job entry subsystem processes the SYSUDUMP as class A output. The system determines the contents of the dump according to one or all of the following:

- Options coded in the SYSUDUMP parmlib member, IEADMP00
- The dump options list for the invoking macro (ABEND, CALLRTM, or SETRP)
- Options entered with the CHNGDUMP operator command

Requesting SYSMDUMP Dumps: To generate a SYSMDUMP, the JCL for the job step must include a DD statement (SYSMDUMP) defining the output data set to hold the dump:

```
//STEPNUMB EXEC PGM=DEFG  
//SYSMDUMP DD DSN=DUMP,DISP=SHR
```

If this step has an abnormal termination, the system takes a machine-readable **SYSMDUMP** and places it in the existing data set named **DUMP**. To print the **SYSMDUMP** you must request the print dump service, (**AMDPRDMP**) to print it. In a TSO environment, you can use **IPCS** to examine the dump at the terminal. Then you can decide whether to print it or not.

The system determines the contents of the **SYSMDUMP** by one or all of the following:

- Options coded in the **SYSMDUMP** parmlib member, **IEADMR00**
- The dump options list for the invoking macro (**ABEND**, **CALLRTM**, or **SETRP**)
- Options entered by the **CHNGDUMP** operator command

SVC Dumps

You use the **SVC** dump to help diagnose errors in system routines. Only an authorized program can request an **SVC** dump. You can request **SVC dumps** (also called **system dumps**) through:

- The **SDUMP** macro as part of **MVS** system component recovery processing,
- The **DUMP** operator command, or
- As an action on a **SLIP** command.

SDUMP processing produces an **SVC** dump. The dump is unformatted, machine-readable and consists of the system data for the address spaces involved with the failing unit of work. The **SVC** dump is stored in a system dump data set on a direct access or tape device. To format and print the **SVC** dump, you can use the print dump service aid, **PRDMP**. In TSO, you can use **IPCS** to format the **SVC** dump and to display it at the terminal.

The options you code on the **SDUMP** macro determine the contents of an **SVC** dump. To modify the options on the **SDUMP** macro, you can use the **CHNGDUMP** command. **IPCS** uses **verbexit** subcommand to invoke these same control statements. Control statements coded for **PRDMP** determine the areas of the **SVC** dump to be printed.

For the details of the **SDUMP** macro, see *SPL: System Macros and Facilities*. For details of the **PRDMP** service aid, see *MVS/XA Service Aids*. For information about **IPCS**, see the *MVS Extended Architecture Interactive Problem Control System (IPCS) Planning and Customization*, *MVS Extended Architecture Interactive Problem Control System (IPCS) User's Guide*, and *MVS Extended Architecture Interactive Problem Control System (IPCS) Command Reference*.

SVC Dump Data Sets

You define the data sets for SVC dumps at IPL, using the **DUMP** parameter in the IEASYSxx member of SYS1.PARMLIB.

After IPL, you can use the the DUMPDS operator command to:

- Define additional system dump (SYS1.DUMP) data sets
- Delete data sets from the list of those defined as available to the system
- Clear system dump data sets.

In MVS/XA, a maximum of 100 data sets can be set aside to hold system dumps. SYS1.DUMP data sets can reside on DASD or tape volumes.

When you define system dump data sets for DASD (DUMP=DASD) or (DASD,xx-yy), you assign a two-digit identifier to the data set (SYS1.DUMPnn). System dump data sets for tape (DUMP=(TA,xxx,yyy)) are identified by the device number (unit address) of the unit on which the tape volume is mounted.

For a detailed discussion of the DUMP parameter on the IEASYSxx member of SYS1.PARMLIB, see *MVS/XA Initialization and Tuning*.

Allocating SYS1.DUMP Data Sets

To prevent an active dump data set from being scratched or taken offline, the system dynamically allocates SYS1.DUMP data sets to the dumping services (DUMPSRV) address space. The dump data sets are allocated with a status of share (DISP=SHR). This means that any user, other than SDUMP, must allocate the data sets as DISP=SHR, or first remove them from system. To remove them from the system, you can issue the DUMPDS DEL command (see “DUMPDS Operator Command”). Even though a tape SYS1.DUMP data set has a status of share, the sequential nature of tape prevents two programs from using it simultaneously.

Suppressing Dumps

Some dumps duplicate previously taken dumps. Processing duplicate dumps is unnecessary and costly. An installation can suppress unnecessary or duplicate dumps through Dump Analysis and Elimination (DAE) or through abend codes. Dump Analysis and Elimination is discussed later in this chapter

Suppressing Dumps Automatically, by Abend Code

Some dumps are almost never needed. The abend code tells you enough about the problem for you to solve it. To reduce unnecessary dumps, you can suppress both ABEND and SVC dumps by specifying **ACTION=NODUMP** on SLIP commands keyed to the self-explanatory abend codes.

In MVS/XA, you can further refine this process by using the SLIP ACTION options that suppress specific *types of dumps* according to the abend code.

If, for example, for abend code B37, you want to suppress SYSUDUMPs but take SYSABEND, SYSMDUMP, SVC dumps, you can code a SLIP command specifying

```
SLIP SET,COMP=B37, ID=XB37, ACTION=NOSYSU, END
```

You can use a member of SYS1.PARMLIB to include selective SLIP commands to suppress dumps initiated by abend codes for which dumps are normally unnecessary. The parmlib member is IEASLPxx. Another member of SYS1.PARMLIB, IEACMD00, as shipped by IBM, contains SLIP commands to suppress:

All types of dumps	(ACTION = NODUMP)
SVC dumps	(ACTION = NOSVCD),
All SVC and SYSUDUMPs	(ACTION = (NOSVC, NOSYSU)), depending on the abend code specified on the COMP = keyword.

You may prefer copying the SLIP commands from IEACMD00 to an IEASLPxx member. IEACMD00 contains restrictions that IEASLPxx does not.

Note: Do NOT use the COMMNDxx parmlib member for SLIP commands. Use IEASLPxx.

See *MVS/XA Initialization and Tuning* for information on the SYS1.PARMLIB members. See *MVS/XA Diagnostic Techniques* for details on using SLIP commands.

Note: If you are not currently using the SLIP facility, you should be aware that the system allocates fixed storage for the SLIP command modules as a result of the execution of IEASLPxx during IPL.

Tailoring ABEND and SVC Dumps: The Dump Options

Using the MVS dumping services, you can control the kinds and amounts of data included in ABEND and SVC dumps. You can control this by manipulating the options that determine the kinds of data included in each type of dump.

Dump options lists are built for each dump type and contain parameters specifying the contents of the dump. The MVS dumping services use the information in the dump options lists to determine the contents of the dumps. According to the options in effect, the dump includes control blocks, data areas, and other information.

Each type of dump has its own dump options list. Each dump option list is built from at least three sources:

- **For ABEND dumps, the sources include:**

1. System default options included in SYS1.PARMLIB members for the different types of ABDUMPs:

For SYSABEND member IEAABD00

For SYSUDUMP member IEADMP00

For SYSMDUMP member IEADMR00

2. Options on a SNAP parameter list indicated by the DUMPOPT parameter on the ABEND, CALLRTM, or SETRP macro. (The DUMPOPT parameter indicates the *list* form of the SNAP macro; the options are coded on the SDATA and PDATA parameters of SNAP, but the SNAP macro is not executed.)
3. Options included on the CHNGDUMP command, which can modify the default parmlib options, or completely override all other options.

- **For SVC dumps, the sources include:**

1. Options supplied by IBM
2. The parameter list provided on the SDUMP macro or the DUMP command
3. Options included on the CHNGDUMP command, which can modify the effective SDUMP options

Figure 17-1 shows the dump options IBM provides – in parmlib members for ABEND dumps, and as automatic options for SVC dumps.

Figure 17-5 shows the system data (SDATA) options that can be included in a dump options list, and what each option produces for ABEND and SVC dumps.

The CHNGDUMP command described in “Tailoring Dumps by Type: The Operator Commands.”

SYSABEND	(IEAABD00)
* SDATA=(CB,LSQA,IO,ERR,DM,SUM,ENQ,TRT) PDATA=(PSW,REGS,SPLS,ALLPA,SA)	
SYSUDUMP	(IEADMP00)
* SDATA=(SUM)	
SYSMDUMP	(IEADMR00)
* SDATA=(NUC,SQA,LSQA,SWA,RGN,SUM,TRT)	
SVC Dump	(SDUMP macro)
SDATA=(ALLPSA,SQA,SUMDUMP)** SUSPEND=NO BUFFER=NO BRANCH=NO QUIESCE=YES	
* Symptom dump is an implicit default; SDATA=NOSYM turns it off. ** As shipped, IEACMD00 contains a CHNGDUMP command that adds the LSQA and TRT options to the SDATA parameter for every SVC dump.	

Figure 17-1. Default Dump Options for ABEND and SVC Dumps

Tailoring Summaries and Symptom Dumps

In many cases, you do not need to see all the data that could be specified in the dump options list for a given dump type. You might rather see a selection of the information being retrieved for the dump.

MVS/XA includes, as part of the dumping services, two ways to look at part of the data retrieved for a dump. These two ways are the **summary dump** and the **symptom dump**.

The summary dump presents the basic data about an error needed to debug most problems. You can add data to it, if necessary.

The symptom dump includes the information and data needed to identify a problem. While the symptom dump may be particularly useful in identifying duplicate dumps, it may be enough to debug some problems.

Summary Dumps

Perhaps the most significant dump option for an MVS/XA installation is the one that requests a summary of the dump contents. Some form of summary is a default option for every type of user or system dump.

The summary dump is most useful as a means of screening out duplicate problems, although it often contains enough information to debug a problem completely.

Figure 17-2 shows the contents of summary dumps for each of the ABEND dump types.

Contents	Dump Type		
	SYSABEND	SYSUDUMP	SYSMDUMP
Dump Title	X	X	X
Abend code and PSW at time of error	X	X	X
Instruction length code	X	X	
Interrupt code	X	X	
ERRORID	X	X	X
Name of active load module, if any	X	X	X
PSW address	X	X	X
Offset into load module of failing instruction	X	X	
System control blocks (ASCB, TCB, RB, LLE, CDE, XL, TIOT, etc.)	X	X	
Error control blocks (RTM2WA, SCB)	X	X	X
Save areas	X	X	
Registers at time of error (from RTM2WA)	X	X	X
Contents of active load module	X	X	
Last previous PRB module	X	X	
2K of storage around PSW and register addresses (see note)	X	X	X
All trace table entries for the dumped ASID	X	X	
Note: This includes only the storage the caller is authorized to access. Duplicate addresses are ignored.			

Figure 17-2. ABEND Summary Dump Contents

Note: The data included in a SYSMDUMP summary dump is the same as that included in the enabled SVC summary dump; see Figure 17-3.

In the case of SYSUDUMP, SUM is the only SDATA option included in its parmlib member (IEADMP00). A SYSUDUMP is frequently requested for problem program debugging, where the error can be traced without reference to many system storage areas. In the parmlib members for SYSABEND (IEAABD00) and SYSMDUMP (IEADMR00), the SUM option is combined with options requesting specific system control blocks and data areas to tailor the dump.

ABEND Summary Dump Formats: When an ABEND summary dump is combined with a dump of other data areas, the summary information is scattered throughout the full dump. When SUM is the only SDATA option for an ABEND dump, the summary information is grouped together and constitutes the entire dump.

SVC Dump Summary Dumps

The SMDUMP SDATA option for SDUMP macro produces three kinds of summary dumps. The contents depend upon the type of SVC dump and the mode of entry to the SDUMP code:

- Disabled, produced by BRANCH = YES, SUSPEND = NO
- Suspend, produced by BRANCH = YES, SUSPEND = YES
- Enabled, produced by BRANCH = NO, SUSPEND = NO

See *SPL: System Macros and Facilities* for the details of coding the SDUMP macro.

Figure 17-3 shows the contents of the three kinds of summary dumps for SVC dump, along with the SDUMP options that produce them.

Contents	Dump Type		
	DISABLED (B = Y) (S = N)	SUSPEND (B = Y) (S = Y)	ENABLED (B = N) (S = N)
ASID record for address space in which dump task is running ¹	X	X	X
SUMLIST/SUMLSTA areas, if requested	X	X	X
PSA,PCCA and LCCA for each functioning processor	X	X	
Current PCLINK stack, ² if present	X	X	
IHSA/XSB/PCLINK stack	X	X	
4K of storage around ⁴ PSW and register addresses from the IHSA	X		
Caller's SDWA, if any	X	X	
4K of storage around ⁴ PSW and register addresses from the SDWA	X	X	
SUPER FRR stacks	X		
WSA vector tables, ³ with save areas they point to	X		
4K of storage around ⁴ address portion of: <ul style="list-style-type: none"> ● I/O old PSW ● Program check old PSW ● External old PSW ● Restart old PSW for each processor	X		
Caller's ASCB		X	
All RTM2WAs pointed to by all TCBs in the address space			X
4K of storage around PSW ⁴ and register addresses from RTM2WAs			X
Current FRR Stack			X
SDUMP's caller: TCB or SSRB		X	
For TCB Mode: All RTM2WAs pointed to by this TCB, with any SDWAs		X	
For SSRB mode: PCLINK stack and SDWA		X	
Register save area storage		X	
Notes. ¹ For disabled and suspend summaries, this is the cross-memory ASID record: home, primary and secondary ASIDS, and the cross-memory lock ASID if a CML is held. ² Pointed to by PSASEL. ³ The global, local and CPU work save area vector tables. ⁴ The addresses are compared, using an address range table, to avoid dumping duplicate areas.			

Figure 17-3. SVC Dump Summary Dump Contents

Note: The data included in the SYSDUMP summary dump (see Figure 17-2) is the same as that in the enabled SVC summary dump.

SVC Dump Summary Dump Formats: The summary information for an SVC dump can be scattered throughout the dump when other data area options are specified with `SDATA = SUMDUMP`. The dumping service writes the machine-readable form of the SVC dump summary to a system dump data set. When you invoke `AMDPRDMP` with its `SUMDUMP` control statement, print dump organizes and formats all the summary data in the dump.

Symptom Dumps

In addition to the summary dump option in their parmlib members, `SYSABEND`, `SYSUDUMP`, and `SYSMDUMP` dumps include, as an implicit default, a 10-line **symptom dump**.

The symptom dump appears as a write-to-programmer message in the job log for the abending job or step. An example of the message is shown in Figure 17-4.

```
IEA995I  SYMPTOM DUMP OUTPUT
ABEND CODE SYSTEM=0C1 TIME=09.08.10 SEQ=00049 CPU=0000 ASID=000B
PSW AT TIME OF ERROR 078D0000 81B000D4 ILC 2 INTC 01
ACTIVE LOAD MODULE=DAESM1 ADDRESS=01B00000 OFFSET=000000D4
DATA AT PSW 01B000CE - 50000F010 000092E7 A05847F0
GPR 0-3 01B000D4 01B02F2C 00000040 01B02F39
GPR 4-7 01B00174 01B02F39 01B02FBC 01B02F38
GPR 8-11 01B02F2C 809FF6D8 01B02EE0 81B00018
GPR 12-15 01B01017 01B02EE0 80F775A0 01B02F39
END OF SYMPTOM DUMP
```

Figure 17-4. Message IEA995I: Symptom Dump Output for SYSABEND and SYSUDUMP

ABDUMP processing always produces the symptom dump output message for a `SYSABEND`, `SYSUDUMP` or `SYSMDUMP`. To eliminate the symptom dump output message, you can specifically turn off the option in the parmlib member. See “Suppressing the Symptom Dump.”

In a TSO environment, you can display the symptom data on the screen by issuing the `PROFILE` command with the `WTPMSG` option.

Note: The symptom output data for a `SYSMDUMP` is put into the dump header record by `SDUMP` at the request of `ABDUMP`. The contents are the same as the symptom data produced for `SYSABEND` and `SYSUDUMP` dumps.

Symptom Data in an SVC Dump

Formatted symptom data appears at the beginning of every SVC dump, in the dump header. This is the same kind of information as that included in the `ABDUMP` symptom dump.

The SVC dump header also includes the system diagnostic work area (SDWA), which contains data retrieved by the recovery routine that requested the dump.

To display symptom data from DASD system dump data sets, you can enter the `TITLE` or `ERRDATA` option on the `DISPLAY DUMP` command. See “Tailoring Dumps by Type: The Operator Commands” for more about the `DISPLAY DUMP` command.

Tailoring Dumps: Other Data Options

Unless the dump options list is limited to the `SUM` or `SUMDUMP` option, the MVS dumping services produce a full dump. The dump consists of all the storage areas and control blocks requested, as shown in Figure 17-5.

To tailor the full dump for each dump, you use the `SDATA` (for system dumps) or `PDATA` options (for user dumps) in the dump options list. The effects of these data options can be divided into two parts:

- Dumping the nucleus
- Dumping specific areas of storage

Dumping the Nucleus

In MVS/XA, the system control code resides in two nucleus areas. One area, in real storage, contains system control modules that never change from one IPL to another. This area is called the DAT-off nucleus. The other area, in virtual storage, is called the DAT-on nucleus. It is divided into two parts:

1. The read-only DAT-on nucleus
2. The read/write DAT-on nucleus

The read-only DAT-on nucleus contains protected address space control modules. The read/write DAT-on nucleus contains modifiable control code.

Note: DAT, dynamic address translation, is the hardware feature that makes virtual storage possible. When DAT is active (DAT-on), you are addressing virtual storage; when it is not active (DAT-off), you are addressing real storage.

The options in MVS dumping services allow you to control which parts of the nucleus are included with each dump. It is seldom necessary to dump all the nucleus modules, or even the entire virtual nucleus, every time an ABEND occurs.

ABDUMP Nucleus Options: `SDATA=NUC|ALLVNUC`: The `ABDUMP` `SDATA` options include two forms of request for a dump of the nucleus:

- `SDATA=NUC` produces only the read/write area of the DAT-on (virtual) nucleus.
- `SDATA=ALLVNUC` produces the entire DAT-on nucleus, including both read/write and read-only areas.

SDUMP Nucleus Options: SDATA=NUC/ALLNUC: The SDUMP SDATA options also produce two kinds of dumps of the nucleus:

- SDATA=NUC produces the read/write area of the DAT-on (virtual) nucleus.
- SDATA=ALLNUC produces a dump of all the modules in both the DAT-off and DAT-on nucleus areas, read/write and read-only.



Dump Option	Data Included in the Dump	Dump Macro/Command Where the Option Can Be Coded		
		SNAP/ABEND Macro	SDUMP Macro	DUMP Command
SDATA = ALL	All possible system data areas for user dump, excluding ALLVNUC	X		
= ALLNUC	All of the nucleus		X	X
= ALLPSA	All prefixed storage areas in the system		X	
= ALLVNUC	All of the DAT-on nucleus, and the PSA	X		
= CB	Formatted control blocks for the task	X		
= CSA	Common service area (Subpools 231, 241)		X	X
= DM	Data management control blocks for the task	X		
= ERR	RTM control blocks for the task	X		
= GRSQ	Global resource serialization queue		X	X
= IO	IOS control blocks for the task	X		
= LPA	Active link pack area modules and SVCs for each ASID (see note)	X	X	X
= LSQA	Local system queue area for each ASID (plus subpools 229,230 for SNAP; 229, 230, 233-35, 253-55 for SDUMP)	X	X	X
= NOALLPSA (NOALL)	Only the PSA for the current processor		X	
= NOPSA	No prefixed storage areas			X
= NOSQA	No system queue area		X	X
= NOSUMDUMP (NOSUM)	No summary dump		X	X
= NUC	Read-only, DAT-on nucleus and PSA, including CVT	X	X	X
= PCDATA	PCLINK stack information	X		
= PSA	Prefixed save areas		X	X
= Q (ENQ)	GRS control blocks for the task	X		
= RGN	Private area of address space, including LSQA and SWA		X	X
= SQA	Entire system queue area (Subpools 227, 228, 239, 245, for SNAP)	X	X	X
= SUMDUMP = SUM	Summary dump (see individual dump types for contents)	X	X	X
= SWA	Scheduler work area (Subpools 236, 237)	X	X	X
= TRT	GTF and system trace data, with variations. Master trace and NIP message hardcopy buffer for SDUMP.	X	X	X
Note: A PDATA option for ABEND dumps.				

Figure 17-5. SDATA Options for MVS/XA Dumps

The DAT-on nucleus is dumped as virtual storage; the DAT-off nucleus is dumped as real storage.

The ABEND dumps do not include the system control areas residing in real storage; the DAT-off nucleus is unlikely to be needed to debug an ABEND dump.

An SVC dump deals with operating system control blocks and data area and can include the DAT-off nucleus. The requestor of an SVC dump must be authorized.

The IEADMR00 parmlib member for SYSMDUMP is the only one that includes dumping any of the nucleus as a default option. One of the IEADMR00 SDATA defaults is NUC. This dumps the read/write section of the virtual (DAT-on) nucleus. If you use the IBM-supplied default options for other ABDUMPS or for an SVC dump, none of the nucleus is dumped.

The data in the DAT-off nucleus does not change. You might want to generate one copy of the DAT-off nucleus to use with other dumps. To save one copy of the DAT-off nucleus, request an SVC dump using the DUMP command. Specify ALLNUC as the only SDATA option.

Note: If you specify SDATA = ALLVNUC for a SYSMDUMP, SDUMP generates the ALLNUC option. All of the nucleus that the caller of SYSMDUMP is authorized to see is dumped, in the caller's key.

Dumping Specific Areas of Storage

The SDATA options other than SUM/SUMDUMP request the dumping of specific system data areas and control blocks.

All MVS dumps but SYSUDUMP include some additional system data areas as default or automatic options. Figure 17-5 shows all the SDATA options for MVS dumps, and the data dumped for each. See Figure 17-1 for the default options for each type of dump.

Additional details about these and other dump options and the dumps they produce can be found in *MVS/XA Supervisor Services and Macro Instructions for SNAP macros* and in *MVS/XA SPL: System Macros and Facilities* for SDUMP and SNAP macros.

To tailor a given type of dump, you change the options in the dump options list. For example, you may wish to suppress the summary dump option for one or more dump types, and the symptom dump message for ABEND dumps. You can make these changes in a number of ways.

Changing the ABEND Summary Dump Option: You can suppress the summary dump for ABEND dumps by deleting the SDATA option, SUM, in the parmlib member. You can also by delete SUM from the dump options list via the CHNGDUMP command.

Suppressing SVC Dump Summary Dumps: You can suppress an SVC dump summary dump by using the CHNGDUMP command to change the SDUMP options list to specify SDATA = NOSUM. See "Tailoring Dumps by Type: The Operator Commands."

Suppressing the Symptom Dump: To prevent the symptom data for an ABEND dump, you can put `SDATA=NOSYM` in the appropriate parmlib member before an IPL, or enter `SDATA=NOSYM` on the `CHNGDUMP` command after an IPL. *MVS/XA Initialization and Tuning* tells how to modify members of `SYS1.PARMLIB`. *MVS/XA System Commands* tells how to use the `CHNGDUMP` command.

Special Parameters and Options for Tailoring Dump Types

To tailor the dumps, you can also use some special options specifically designed to produce more meaningful dumps of the data requested.

Dumping Program Data Areas in ABEND Dumps: In ABEND dumps, you can request dumps of data areas specific to the abending program (task). To do this, you can use the `PDATA` parameter on the `SNAP` parameter list.

Besides the default `PDATA` options included in `IEAABD00`, for `SYSABEND` dumps (see Figure 17-1), you can request a dump that includes all the subtasks belonging to the abending task. The option that provides this data is `SUBTASKS`. `SUBTASK` is not supplied as a default option. To add `SUBTASKS` – or any other valid options – to the options list for an ABEND dump, either change the entries in the appropriate parmlib member or use the `CHNGDUMP` command.

Including Storage Subpools in ABEND and SVC Dumps: You can include storage subpools in both ABEND and SVC dumps. For ABEND dumps, you request *specific* subpools using the `SUBPLST` parameter. With `SUBPLST`, you can point to a list of the subpool numbers you want dumped. If you do not code `SUBPLST`, the `SPLS` option causes the dumping of all the virtual storage subpools. `SPLS` is one of the default `PDATA` options for `SYSABEND` dumps.

For an SVC dump, you use the `SUBPLST` option, specifying a list of subpool numbers, along with address space identifiers (ASIDs), that you want dumped. If you code the `KEYLIST` option with the SVC dump `SUBPLST` parameter, the dump is restricted to the subpool areas that have the storage protection keys specified with the `KEYLIST` option.

Neither of these options is a default for an SVC dump; you must add it to the options list on the `SDUMP` macro.

Note that the SVC dump summary dump does not include any storage subpools. If you want any storage subpools dumped, you must include `SUMLIST` or `SUMLSTA` in the options list and specify area you want.

Tailoring Dumps by Type: The Operator Commands

In *MVS/XA*, four operator commands are useful for controlling dump output. They are:

1. `CHNGDUMP` (or `CD`), to dynamically alter dumping options.
2. `DISPLAY DUMP` (or `D D`), to determine current dumping options or system dump data set status, title and error data.

3. DUMPDS (or DD), to dynamically alter the list of system dump data sets, or to clear the system dump data sets.
4. DUMP, to take a system dump under the operator's control.

CHNGDUMP Operator Command

You can use the **CHNGDUMP** command to

- Dynamically alter the dump options list for a given dump type,
- Add new options to the dump options list,
- Change the mode of the dump, or
- Suppress a dump.

CHNGDUMP accepts most of the SDATA dump-tailoring options available for ABEND and SVC dumps, and some of the PDATA options for ABEND dumps.

Using the CHNGDUMP operator command, you can alter the makeup of any dump to suit the requirements of a given problem. See *MVS/XA System Commands* for all the CHNGDUMP options. Of particular interest in tailoring dumps are the following:

- For ABDUMPS (SYSABEND, SYSUDUMP, SYSMDUMP):
 - ALLNUC (for SYSMDUMP only) requests the dumping of the entire nucleus, including the DAT-off nucleus.
 - ALLVNUC (for SYSABEND and SYSUDUMP) requests the dumping of the entire DAT-on (virtual) nucleus, read/write and read-only.
 - NUC requests the dumping of the read/write area of the DAT-on nucleus.
 - NOSYM suppresses the symptom dump message.
 - SUBTASKS (PDATA option for SYSABEND and SYSUDUMP) requests the dumping of the program data of all the subtasks of the task requesting the dump.
 - SUM requests the summary dump (the default).
- For SVC Dumps (SDUMP):
 - ALLNUC (SDATA option) requests the dumping of the entire nucleus, DAT-off and DAT-on, read-only and read/write.
 - NUC requests the dumping of the read/write area of the DAT-on nucleus.
 - SUM requests the summary dump (the default).
 - NOSUM suppresses the summary dump.

Using the NODUMP mode on the SET operand, you can suppress a given dump type completely.

DISPLAY DUMP Operator Command

Among the many kinds of information available via the DISPLAY command is current dump information. When the operator enters **DISPLAY DUMP** or **D D**, followed by one of the four parameters, the requested information appears on the screen, as follows:

Command Syntax	Result
D D,OPTIONS	Dump mode and dump options currently in effect for ABEND and SVC dumps.
D D,STATUS	A summary of the status (full or available) of the SYS1.DUMP data sets (DASD and tape) currently defined to the system. (This is the default.)
D D,TITLE,DSN=(aa,bb-cc)	Dump titles for the full DASD dump data sets named in DSN=. "aa," "bb," and "cc" correspond to the two-digit suffixes from the SYS1.DUMPnn data set designations.
D D,ERRDATA,DSN=nn	Error data for the full DASD dump data sets named in DSN=. "nn" corresponds to the two-digit suffix from the SYS1.DUMPnn data set designation.

STATUS is the default parameter on the DISPLAY DUMP command.

The STATUS summary appears in two parts; one is for DASD dump data sets, the other for tape.

TITLE and ERRDATA can be used only for system dump data sets residing on direct access devices. They are mutually exclusive; ERRDATA displays the same information as TITLE, plus the following:

- Error identifier
- Abend code
- Name of the failing module
- Name of the failing CSECT
- Time of error
- Processor identifier
- PSW at the time of error
- Registers at the time of error

Note that this information is a subset of the information supplied in the symptom dump; it is taken from the dump header record.

The DSN= parameter allows you to restrict the display of title and/or header record information to the DASD dump data sets you specify as two-digit numbers in the range 00 to 99.

You can ask to see TITLE or ERRDATA for individual data set numbers, or for a range of data set numbers, or for all of your full DASD dump data sets.

Note: The parameters ERRDATA, OPTION, STATUS, and TITLE are mutually exclusive. See *MVS/XA System Commands* for more information about the DISPLAY DUMP command.

DUMPDS Operator Command

You can use the dump data set (DUMPDS or DD) command to modify the system queue of available dump data sets without having to re-IPL the system.

The DUMPDS command has three possible operands: ADD, DEL (delete), and CLEAR; plus the DSN and UNIT specifications.

You identify the data sets you want to add to or delete from the system dump data set queue by DASD data set number (DSN=nn) or by tape device number (UNIT=uuu). The two-digit number for DASD dump data sets is the two-digit suffix from SYS1.DUMPnn (00-99).

Adding System Dump Data Sets

The DUMPDS ADD function adds system dump data sets to the queue of those defined as available for SVC dumps.

Note: You must pre-allocate and catalog any DASD data set you want to add to the dump data set queue, before issuing the DUMPDS ADD command. It is also a good idea to provide protection (password or RACF) for the data set, because system dumps contain sensitive data.

Deleting System Dump Data Sets

The DUMPDS DEL,DSN= function removes DASD dump data sets, singly or as a group, from the queue, so they can no longer be accessed by SDUMP or the DISPLAY DUMP command processor. The deleted DASD data sets are not scratched or uncataloged. DUMPDS DEL,DSN=ALL removes all the DASD dump data sets from the queue; you may also specify a range of DASD data set numbers, as in DSN=(xx-yy).

DUMPDS DEL,UNIT=uuu removes tape data sets from the system queue one at a time; the system unloads the tape volume and deallocates the device.

Clearing Dump Data Sets

Having used the DISPLAY DUMP command (see “DISPLAY DUMP Operator Command”) to display the status, options, and error data for the full system dump data sets associated with a processor, the operator can issue the DUMPDS command to clear the data sets whose dumps have already been off-loaded or whose contents reflect duplicate problems.

DUMPDS CLEAR,DSN=nn clears a DASD system dump data set; DUMPDS CLEAR,UNIT=uuu does the same for tape.

DUMPDS CLEAR makes full dump data sets available to SDUMP for other dumps. The system puts an end-of-file marker at the beginning of the specified data set, in effect pronouncing it empty and available.

As in the DEL parameter, DASD and tape data sets are handled differently for the CLEAR parameter. DUMPDS CLEAR,DSN=ALL clears all your currently-defined full DASD dump data sets at once, and DUMPDS CLEAR,DSN=xx-yy clears the DASD dump data sets whose identifiers are included in range xx-yy. You must clear each tape dump data set individually, by device number (DUMPDS CLEAR,UNIT=uuu).

DUMP Operator Command

Using the DUMP command, you can request and tailor a system dump, directing it to a system dump data set. After entering the DUMP command and the dump title, you receive message IEE094D, which asks for operands to define the dump. Using the REPLY (R) command, you supply the operands.

In MVS/XA, the SDATA parameter for the DUMP command can include the ALLNUC option, which requests the dumping of the entire nucleus. See “Dumping the Nucleus” for a description of the logical parts of the MVS/XA nucleus.

You can specify certain system storage areas by name on the SDATA parameter of the DUMP command, as in the SDUMP macro, or you can specify areas of storage by address, on the STOR parameter. The STOR parameter requires the beginning and ending addresses of an area of storage, specified as 4-byte hexadecimal numbers (040BA040) or 7-digit decimal numbers followed by “K” (0150860K).

You can choose to designate storage as addresses or system areas or address spaces (ASID parameter); thus, you can tailor the dump to your problem-solving needs.

See *MVS/XA System Commands* for more information on the DUMP command, and *MVS/XA System Messages* for the texts of the messages involved with its execution.

Tailoring and Suppressing Individual Dumps: The User Exits

In MVS/XA, you can use your own exit routines to analyze a dump – before or after it has been taken – and decide if it should be taken, modified, formatted, printed, or terminated.

The dump-tailoring exit for user (ABDUMP) dumps occurs before a dump is taken; the exit for system (SDUMP) dumps occurs after a dump has been taken.

In both cases, you can supply your own routines by putting their names in exit tables provided by IBM. The dumping services component invokes your pre- or post-dump exit routines as appropriate.

Pre-Dump Exits for User Dumps

You specify pre-dump exit routines in the SNAP/ABDUMP exit table, load module IEAVTABX. Each exit routine has an 8-byte entry in the exit table.

Module IEAVTABX, as shipped by IBM, has the following format: a count field at offset zero and 10 null entries (8 bytes of X'40'), followed by the module ID and an end-of-list indicator (8 bytes of zeros).

You fill in the exit entries with the names of your own load modules, in the order you want them invoked during ABEND dump processing. The count field shows how many module names are in the table.

If you do not want an exit routine to run, you overlay its module name with blanks, and decrease the number in the count field.

Your process of adding or deleting entries involves assembling the module names — and blank entries — and link editing them as IEAVTABX. The modules named in the exit table must reside in an authorized system library.

It is probably best to change the exit table prior to a cold-start IPL, so the exits will be in place for recovery management as soon as the system is up.

The pre-dump exits work much the same way as all MVS/XA user exits: your routine receives control in supervisor state, key 0, enabled, unlocked in non-cross-memory mode, and in 31-bit addressing mode.

When your exit is given control, the contents of the registers are as follows:

- register 1 points to a parameter list (mapped by mapping macro IHAABEPL)
- register 13 points to a save area
- register 14 contains the return address
- register 15 contains the entry-point address.

Your routine must use standard entry and exit linkages, and must place a return code in register 15 before returning to its caller. The possible return codes are:

Code	Meaning
0	Continue processing with existing dump options
4	Continue processing with new options
8	Terminate the dump

As you can see from the return codes, you can modify or suppress a dump, using a pre-dump exit routine.

Note: If your exits contain macros, the macros must be compatible with MVS/XA. See the discussion of the SPLEVEL macro in *MVS/XA SPL: System Macros and Facilities*.

The parameter list whose address is in register 1 when your exit receives control includes:

- Jobname for the abending task
- Abend code

- Address of the SDWA for the error, if one exists
- Name of the active load module, if any
- Pointer to the SNAP parameter list (dump options) to be updated by the exit routine
- Parameter list level indicator
- Return code from the previous exit, if any

See the *MVS/XA Debugging Handbook* for the mapping of IHAABEPL.

Post-Dump Exits for System Dumps

At the end of each SVC dump and SYSMDUMP, the dumping services component gives control to the routines named in data CSECT IEAVTSEL for post-dump processing. You put exit routine module names in IEAVTSEL, which resides in SYS1.LINKLIB, either by reassembling the module or by means of the AMASPZAP service aid.

As shipped by IBM, IEAVTSEL contains 10 8-byte fields of blanks (X'40') separated by 4-byte reserved flag fields. The end of the list is indicated by 12 bytes of zeros.

You can put your own exit module names in the exit list, and you can delete entries by overlaying them with blanks.

Your exit routines are given control in the order in which their names appear in the exit list. When a routine gets control, the contents of the registers are as follows:

- register 1 points to a parameter list, mapped by IHASDEPL
- register 13 points to a save area
- register 14 contains the return address
- register 15 contains the entry point address

The exit routines must use standard entry and exit linkages.

The exit routines receive control in the addressing mode specified in their link edits, in supervisor state, in key 0, in task mode, in the DUMPSRV address space.

Before an exit routine returns control to its caller, it must put a return code into register 15. Any nonzero return code indicates that the exit was unsuccessful.

The parameter list passed to the exit routine is mapped by macro IHASDEPL, in SYS1.MACLIB. SDEPL includes the following:

- Exit status flags
- Address of the dump header record mapped by the AMDDATA mapping macro

- Address of a 200-byte exit work area
- Address and length of an exit interface work area

The two exit status flags indicate if errors occurred in previous exits.

The first, SDEPLEXE, is turned on when an error occurs in an exit routine, and then turned off if a later exit completes successfully. It indicates that the immediately preceding exit had an error.

The second flag, SDEPLERR, is turned on when an error occurs, and is not turned off until all the exits have been called. It indicates that some preceding exit had an error.

The copy of the dump header record contains the following kinds of data:

- Dump type (SVC dump, SYSMDUMP, SVC dump for a SLIP request)
- Dump data set name (SYS1.DUMPnn or, for SYSMDUMP, a unique name)
- Error identifier from this dump
- A copy of the SDWA of the caller of SDUMP (except for SLIP dumps)
- Symptom data

See the mapping macro AMDDATA in the *MVS/XA Debugging Handbook* for other information contained in the dump header.

The exit work area is a general work area for the exits. It is cleared to zeros before each exit gets control.

You can use the exit interface area as a communication area, where one exit can pass information to successive exits, or as a work area. It is only cleared to zeros before the first exit gets control.

Post-dump exits can be used to automate dump processing, to extract and log information from the dump header record, to locate duplicate problems, and to suppress printing of unnecessary dumps.

Using a Post-Dump Exit to Analyze and Handle System Dumps

Following is a list of steps you might take to recognize and note duplicate problems in system dumps, and suppress printing of the dumps you do not need.

- Extract a set of symptoms from the header record sufficient for matching duplicate problems; for example:
 - Abend code
 - Reason code
 - Module name
 - Smallest difference between the PSW address and a register address at the time of error
 - Other indicators of the source of an error
- Build these symptoms into a *symptom record*.

- Dynamically allocate a log data set (SYS1.DUMPLOG, for example), which contains all the symptom records from previous dumps, along with status fields to keep track of the number of times each symptom record has occurred.
- OPEN the dump log data set for input, or update it in place.
- Read each record in the dump log and compare it to the symptom record for the current dump.
- If the current record matches a previous dump symptom record, the problem is a duplicate, and the dump data set can be cleared.
- If no match is found for the current symptom record, your routine can notify the operator to start a job to print or off-load the dump data set.
- Add the current symptom record to the end of the SYS1.DUMPLOG data set (DISP = MOD).
- Close the dump log data set and dynamically deallocate it.
- Use WTO to add a record of the exit's action to the console log.

Note: These actions duplicate some of the processing that can be performed by the dump analysis and elimination feature of MVS/XA; see the following discussion of DAE. If DAE is installed and operational on your system, you probably do not need to use post-dump exits to compare symptoms from SVC or SYSMDUMP dumps.

Processing Machine Readable Dumps

To process machine readable dumps, you can use IPCS or PRDMP. To tailor IPCS and PRDMP to your needs, you can modify the following areas:

1. BLSCECT parmlib member
2. IPCSPRxx parmlib member
3. IPCS dialogs

BLSCECT: Use this parmlib member to define dump analysis and formatting exits that execute under IPCS and PRDMP. If you have written your own dump exits for your installation, you must modify the BLSCECT member to define the exit to IPCS and to PRDMP.

Individual users may specify a substitute parmlib member using the IPCSPARM DD statement. See *MVS/XA Initialization and Tuning* for information on the BLSCECT member of SYS1.PARMLIB.

IPCSPRxx: You can use this member to define session default parameters for all IPCS users on the system. See *MVS/XA Initialization and Tuning* for information on the IPCSPRxx member of SYS1.PARMLIB.

IPCS dialogs: To assist in the debugging process, IPCS provides a full-screen dump analysis. You can modify the IPCS dialog to your installation's needs. See *MVS/XA Interactive Problem Control System Planning, and Customization* for more information about IPCS dialog and options.

Dump Analysis and Elimination (DAE)

Dump analysis and elimination (DAE) allows an installation to suppress SYSMDUMP and SVCDUMP dumps that are unnecessary because they duplicate previously taken dumps. DAE operates as part of SDUMP processing; it performs the following functions:

- **Matching**--DAE compares the symptoms from an SVCDUMP or SYSMDUMP to symptoms of dumps previously recorded in a special system data set (SYS1.DAE).
- **Updating**--If the DAE parameter record specifies **UPDATE**, DAE either updates the incidence count in SYS1.DAE when a duplicate dump occurs, or creates a new record in SYS1.DAE for a unique dump not previously recorded.
- **Suppressing**--DAE prevents a dump from occurring when the DAE parameter record specifies the **SUPPRESS** option, the dump's symptom data matches the symptom data of a duplicate dump already recorded in SYS1.DAE, and the VRADAE key is set.

The sources of data for DAE processing are:

- The DAE parameter record in ADYSETxx members of SYS1.PARMLIB, containing DAE processing options
- The ADYDFLT module, containing default values for DAE processing
- The SYS1.DAE data set, containing symptom data that DAE uses
- The ABDUMP symptom area of the dump header record (AMDDATA)
- The system diagnostic work area (SDWA), which DAE uses to locate symptom data for the dump

The purpose of this section is to describe how you can modify DAE to fit the needs of your installation. To achieve this purpose, this section defines several terms used in a specific context by DAE, describes the input data sources to DAE, and discusses DAE processing.

Definitions

This section defines some terms useful in understanding how DAE analyzes and suppresses dumps.



Symptoms


DAE defines a symptom as problem data that is useful in explicitly defining the failing state of the system. A DAE symptom has two parts: a keyword, describing the type of symptom, and the actual symptom data. For example, DAE uses the system abend completion code, such as X'00C4', as symptom data. DAE attaches the system abend completion code keyword, AB/S, to the abend code. The resultant symptom is AB/S00C4. See "Keys and Keywords" for definitions of keys and keywords.

Symptom Strings

DAE defines a set of symptoms as a symptom string. The two types of DAE symptom strings are MVS and RETAIN. If an MVS symptom string meets the criteria for uniquely identifying a problem and also matches the symptom string from a previous dump, DAE considers the dump to be a duplicate of the original dump.

Symptom Queue

DAE creates an in-storage symptom queue from selected records in the SYS1.DAE data set. Each record in SYS1.DAE contains symptom strings from previously recorded dumps. DAE uses the in-storage symptom queue when matching for duplicate dumps.




The in-storage symptom queue is created when the **SET DAE=00** operator command is executed, or whenever DAE is made operational. When the **SET DAE=01** operator command is executed, and whenever DAE is made non-operational, DAE frees the storage occupied by the symptom queue. DAE does the following:

- When DAE finds a problem that is a duplicate of a problem described in the in-storage symptom queue, DAE increases the incidence count in the symptom queue element by one. If the DAE parameter record specifies **UPDATE**, DAE increases the occurrence count in SYS1.DAE. Then, if all criteria for dump suppression are present, DAE suppresses the dump.
- When DAE finds a unique problem, it adds an entry to the in-storage symptom queue to describe this new problem. If the DAE parameter record specifies **UPDATE**, DAE adds a new record to SYS1.DAE. Then DAE allows dumping to continue.

Keys and Keywords

A key is a hexadecimal number that represents a symptom. A visible keyword is a printable EBCDIC identifier for a symptom. Every key has a corresponding visible keyword. DAE uses these visible keywords to create symptoms.

DAE uses two types of visible keywords: MVS and RETAIN.



MVS Keywords

DAE uses MVS visible keywords to create MVS symptom strings. MVS symptom strings contain visible keywords meaningful to MVS users, such as MOD/ for the module name and CSECT/ for the CSECT name. DAE requires that MVS symptom data be at least one byte long; the maximum length is 50 bytes. These 50 bytes include the visible keyword, the slash separator, and the symptom data. If the MVS symptom is too long, DAE truncates the symptom at 50 bytes.

The maximum length of the MVS symptom string is 150 bytes. If the MVS symptom string is too long, DAE truncates the symptom string at 150 bytes.

See “MVS Symptoms” for a discussion of how DAE creates MVS symptoms. *MVS/XA Debugging Handbook, Volume 1* includes a table showing all DAE keys and keywords, and their meanings.

RETAIN Keywords

DAE uses RETAIN visible keywords to create RETAIN symptom strings. RETAIN symptom strings contain visible keywords that are acceptable for RETAIN searches. For RETAIN symptoms, the maximum length is 15 bytes. These 15 bytes include the visible keyword, the slash separator, and the symptom data. If the RETAIN symptom is too long, DAE truncates the symptom at 15 bytes.

See “RETAIN Symptoms” for a discussion of how DAE creates RETAIN symptoms. *MVS/XA Debugging Handbook, Volume 1* includes a table showing all DAE keys and keywords, and their meanings.

Minimum Symptoms

DAE requires a minimum set of symptoms for MVS symptom strings before it matches for duplicate dumps. A minimum set of symptoms is the smallest amount of symptom data that DAE needs to successfully process the matching function. DAE requires the following:

- A minimum of five non-null symptoms in the MVS symptom string. Each symptom must contain a meaningful identifying element, such as module name, CSECT name, abend code, etc.
- A minimum MVS symptom string length of 25 bytes.
- A set of required symptoms and optional symptoms. Each MVS symptom string must contain all required symptoms, and enough optional symptoms, to meet the minimum count of five non-null symptoms.

Required Symptoms

DAE requires certain symptom data before it matches for duplicate dumps. If the required data is not present, DAE does not perform matching. DAE defines a list of default required symptom keys in module ADYDFLT. See “ADYDFLT” for a description of these defaults.

Optional Symptoms

Optional symptoms assist in DAE match processing. DAE uses only nonzero and non-blank optional symptoms. DAE does not require the presence of optional symptoms; but if optional symptoms are present, DAE uses them for matching. DAE defines a list of default optional symptom keys in module ADYDFLT. See “ADYDFLT” for a description of these defaults.

Input to DAE

DAE uses several input data sources for its processing; they are:

- DAE parameter record, a set of DAE processing options that is contained in an ADYSETxx member of SYS1.PARMLIB
- ADYDFLT, a non-executable load module containing default symptom keys and minimum matching criteria
- SYS1.DAE, a data set containing a record for every unique dump for which DAE has recorded information
- The ABDUMP symptom area of the dump header record, which is mapped by the AMDDATA macro
- SDWA, the system diagnostic work area, and the variable recording area (VRA) in the SDWA, from which DAE extracts symptom information

DAE Parameter Record in SYS1.PARMLIB

The DAE parameter record is in an ADYSETxx member of SYS1.PARMLIB. This parameter record specifies the options for DAE processing. IBM supplies three ADYSETxx members of SYS1.PARMLIB for DAE: ADYSET00, ADYSET01, and ADYSET02. Another SYS1.PARMLIB member, IEACMD00, specifies the automatic startup of DAE.

You can create additional ADYSETxx members of SYS1.PARMLIB, so that each parameter record has a combination of DAE options that your installation might need. The **SET DAE=xx** operator command references an ADYSETxx member of SYS1.PARMLIB. The operator uses the **SET DAE** command to change the operational mode of DAE.

MVS/XA Initialization and Tuning describes each IBM-supplied ADYSETxx member of SYS1.PARMLIB, and discusses the syntax of the DAE parameter record. *MVS/XA Operations: System Commands* describes the syntax and use of the **SET DAE** operator command.

The format of the DAE parameter record in any ADYSETxx member of SYS1.PARMLIB is as follows:

```
DAE={STOP | START [ , RECORDS (n) ] [ , SVCDUMP (p) ] [ , SYSMDUMP (p) ] }
```

Figure 17-6. Format of DAE Parameter Record

None of the keyword parameters is positional. The description of the parameters and subparameters (*p*) is in *MVS/XA Initialization and Tuning*.

IEACMD00

The IEACMD00 member of SYS1.PARMLIB contains the operator command **SET DAE=00**. IEACMD00 executes before any other COMMNDxx parmlib member, and causes DAE to read and interpret the ADYSET00 member of SYS1.PARMLIB. Because ADYSET00 contains **DAE=START** and DAE processing options, DAE is automatically started during system initialization.

Note: If SYS1.DAE does not exist, automatic start-up of DAE fails.

SET DAE Operator Command

The **SET DAE** operator command references a corresponding ADYSETxx member of SYS1.PARMLIB. The ADYSETxx member contains a DAE parameter record that starts or stops DAE processing. The **SET DAE** command has the following format:

```
SET DAE=xx
```

xx is a two-digit alphanumeric character.

In addition to the IBM-supplied ADYSETxx members of SYS1.PARMLIB, you can create ADYSETxx members for specific DAE processing options. The operator can use the **SET DAE** command to change the operational mode of DAE.

ADYDFLT

ADYDFLT is a non-executable load module containing the constants that define the minimum criteria for DAE matching. ADYDFLT also contains the default keys that DAE uses for match processing. The ADYDFLM mapping macro maps ADYDFLT; *MVS/XA Debugging Handbook* describes ADYDFLM and the structures that it maps.

As supplied by IBM, ADYDFLT contains a list of both required symptom keys and optional symptom keys. Figure 17-7 shows the required keys; Figure 17-8 shows the optional keys.

IHAVRA Label	Symptom
EFLDMD	Name of the failing load module
EFC SCT	Name of the failing CSECT

Figure 17-7. Required Symptom Keys in ADYDFLT

IHAVRA Label	Symptom
EFABS	System abend code
EFABU	User abend code
EFREXN	Name of CSECT containing the recovery routine
E1FI1C	Failing instruction area
EFPSW	PSW/register difference
E1HRC1C	Reason code for this ABEND
E1PIDS1C	Product/Component identifier
E1SUB1C	Subfunction name

Figure 17-8. Optional Symptom Keys in ADYDFLT

MVS/XA Debugging Handbook, Volume 1 includes a table showing all DAE keys and keywords, and their meanings.

DAE creates the required and optional symptoms by obtaining symptom data from the SDWA and the ABDUMP symptom area, and concatenating the symptom data to the appropriate symptom keyword.

For a required symptom, DAE must locate the data to create the symptom. If DAE cannot locate the data for a required symptom, or if the data is all blanks or all zeros, DAE cannot build that required symptom. The absence of a required symptom prevents DAE from matching for duplicates.

For the optional symptoms, DAE locates the data to create the symptom. If DAE cannot locate the data, or if the data is all blanks or all zeros, DAE cannot build that optional symptom. The absence of an optional symptom does not prevent DAE from matching for duplicates, as long as minimum requirements for matching are met. Refer to “Minimum Symptoms” for a discussion of minimum MVS symptom string requirements.

Changing Symptom Keys in ADYDFLT

You can change the list of symptom keys defined in ADYDFLT. In addition, a recovery routine can dynamically add to the list of symptom keys defined in ADYDFLT by using the VRADATA macro.

Searching for Symptom Data

DAE creates symptoms by obtaining symptom data from the SDWA and the ABDUMP symptom area, and concatenating the symptom data to the appropriate symptom keyword. When DAE searches for the symptom data to concatenate to a symptom keyword, it follows certain rules:

- DAE searches for the required and optional symptom data in the order of the symptom keys as shown in Figure 17-7 and Figure 17-8.
- If your recovery routine adds required symptom keys (by specifying VRAREQ on the VRADATA macro) they follow the required symptom keys already defined in ADYDFLT. DAE first searches for symptom data for the ADYDFLT-defined required symptom keys, and then searches for symptom data for any user-defined required symptom keys.
- If your recovery routine adds optional symptom keys (by specifying VRAOPT on the VRADATA macro), they follow the optional symptom keys already defined in ADYDFLT. DAE first searches for symptom data for the ADYDFLT-defined optional symptom keys, and then searches for symptom data for any user-defined optional symptom keys.

SYS1.DAE

SYS1.DAE is a data set that contains symptom records for uniquely identified dumps processed by DAE. DAE creates or updates a record in SYS1.DAE when the DAE parameter record specifies **UPDATE**. You can browse the symptom records in SYS1.DAE if you allocate the SYS1.DAE data set **DISP=SHR**. DAE requires SYS1.DAE to function; if there is no SYS1.DAE data set created from a previous IPL, you must create a new one.

Creating the SYS1.DAE Data Set

You can create a SYS1.DAE data set after system generation using IEFBR14 to allocate space for SYS1.DAE. Note that you can use TSO or ISPF facilities to allocate space for SYS1.DAE, and to catalog it.

The DAEALLOC member of SYS1.SAMPLIB contains JCL that an installation can execute to create the SYS1.DAE dataset. Figure 17-9 shows the JCL in this SAMPLIB member.

```

//ALLOCDAE JOB MSGLEVEL=(1,1)
//DAEALLOC EXEC PGM=IEFBR14
//*
//*      DUMP ANALYSIS AND ELIMINATION (DAE) SC143
//*
//*      SAMPLE JCL TO ALLOCATE THE DATASET USED BY DAE TO
//*      CONTAIN THE RECORD OF ALL DUMPS THAT IT HAS IDENTIFIED.
//*      THIS EXAMPLE ASSUMES A 3350 DEVICE AND THAT FULL TRACK
//*      BLOCKING IS THE MOST DESIRABLE. DAE REQUIRES FIXED
//*      LENGTH 255 BYTE RECORDS. OTHER PARAMETERS SHOULD BE
//*      CHOSEN BY CONSIDERING WHAT IS BEST FOR THE INSTALLATION.
//*
//DD1      DD DSN=SYS1.DAE,DISP=(NEW,CATLG),
//          DCB=(RECFM=FB,DSORG=PS,LRECL=255,BLKSIZE=18870),
//          SPACE=(TRK,(6,2)),UNIT=3350
//*
//*      IN THIS EXAMPLE THE BLKSIZE OF 18870 YIELDS 74
//*      RECORDS PER TRACK. THEREFORE THE PRIMARY ALLOCATION
//*      WILL BE 440 RECORDS. THIS IS A GOOD NUMBER TO START
//*      WITH IF NO SPECIFIC EXPERIENCE IS AVAILABLE.
//*

```

Figure 17-9. Sample JCL in SYS1.SAMPLIB for Creating SYS1.DAE

Using an Existing SYS1.DAE Data Set

When you use an existing SYS1.DAE data set, it should have the following DCB attributes:

```

RECFM=F or RECFM=FB
LRECL=255
DSORG=PS

```

Note that record blocking provides the most efficient use of a direct access device. Also, you should allocate enough space on the DASD for about 400 records of 255 bytes each.

Your SYS1.DAE data set should be password- or RACF-protected. It must be cataloged, and cannot be shared across systems.

Updating SYS1.DAE After Dump Processing

To complete its processing, DAE updates the SYS1.DAE data set if the DAE parameter record specified **UPDATE**. If DAE determines that the current dump is a unique dump, DAE creates a new record in SYS1.DAE to describe the dump. If DAE determines that the current dump is a duplicate of a previous dump, DAE updates the first matching record in SYS1.DAE that describes the original dump.

Notes:

1. *If DAE finds more than one record in SYS1.DAE that describes the same dump, DAE only uses and updates the first record it finds. DAE adds this record to the in-storage symptom queue.*

2. If DAE determines that the existing record in *SYS1.DAE* for the most recent occurrence of this dump is older than 180 days, DAE does not add the record to the in-storage symptom queue.

ABDUMP Symptom Area of the Dump Header Record

DAE extracts symptom data from the ABDUMP symptom area of the dump header record. *AMDDATA* is the macro that maps the data in the dump header record. DAE uses two of the structures defined by this macro: *PRDSYSMD* and *ADSSRNSD*.

PRDSYSMD is the ABDUMP symptom area. DAE extracts symptoms from *PRDSYSMD* symptom data relevant to *SYSMDUMPs*. Refer to “Symptom Dumps” for more information about the ABDUMP symptom area.

ADSSRNSD is the area in which DAE stores symptom strings and other processing data for the dump header record. The dump formatting programs (Print Dump and IPCS) later format and print the information stored in *ADSSRNSD*.

SDWA

DAE searches various fields in the SDWA to locate symptom data it uses to create symptoms. Specifically, DAE uses these areas:

- The fixed portion of the SDWA
- The first recordable extension of the SDWA
- The second recordable extension of the SDWA
- The variable recording area (VRA) in the SDWA

MVS/XA Debugging Handbook describes the SDWA.

DAE Processing

This section describes DAE processing, including:

- DAE initialization
- Symptom extraction
- How DAE creates symptoms
- Criteria for DAE to match for duplicates
- Criteria for DAE to suppress dumps
- DAE **SUPPRESS** and **UPDATE** processing
- The dump header record
- Overrides to DAE suppressing dumps

DAE Initialization

DAE is started at system initialization by the **SET DAE=00** command in the SYS1.PARMLIB member, IEACMD00. The DAE function that reads and updates SYS1.DAE, and the DAE post-dump exit, reside in the DUMPSRV address space. If the DAE parameter record specifies or implies matching, DAE reads the SYS1.DAE data set. Then DAE selects certain records from SYS1.DAE, and creates an in-storage symptom queue. DAE loads the default options module, ADYDFLT. If the operator enters any **SET DAE** commands, DAE interprets and processes them.

Symptom Extraction

When SDUMP receives control to take a dump, SDUMP calls DAE, and passes it information that DAE needs to determine if it should suppress the dump. DAE initializes the DAE symptom extraction (DSX) parameter list with the addresses of these source data areas:

- The fixed portion of the SDWA
- The first recordable extension of the SDWA
- The second recordable extension of the SDWA
- The variable recording area (VRA) of the SDWA
- The ABDUMP symptom area (in the dump header record)

DAE creates the MVS and RETAIN symptoms from the symptom data in the SDWA, the SDWA extensions, and the ABDUMP symptom area. DAE places the symptoms into the DSX. Each entry in the DSX contains data from one symptom found by DAE. Then, DAE builds a symptom string from all the entries in the DSX, and places the symptom string in the dump header record.

The keywords in a RETAIN symptom string are not as specific as the keywords in an MVS symptom string. In constructing the MVS symptom string, DAE extracts symptoms from specific areas that are meaningful to the MVS specialist. Thus, the DAE symptom string is designed to have diagnostic meaning for MVS specialists. The RETAIN symptom string is designed for those who are accessing RETAIN, not necessarily MVS specialists. Most RETAIN users use the RETAIN symptom string to determine if a problem is already recorded in RETAIN, and not for diagnostic purposes.

How DAE Creates Symptoms

In designing recovery routines that use or create symptoms, you should know how DAE processes data to create symptoms. Your recovery routine must create symptom data that yields definitive, reproducible, and understandable symptom strings. DAE uses certain editing rules to create symptoms, depending on the type of symptom (MVS or RETAIN) and the type of source data defined in the SDWA and the ABDUMP symptom area.

MVS Symptoms

DAE creates MVS symptoms according to these rules:

- The format of a symptom is *keyword/data*.
- The length of the *keyword* is variable, and the length of the *data* is variable.
- The maximum length of the symptom is 50 bytes, including the keyword, the slash (/) separator, and the symptom data.
- The final character in the symptom cannot be the slash separator.

In addition, DAE follows certain guidelines for different types of source data.

Hexadecimal Source Data: DAE creates MVS symptoms from hexadecimal source data according to these rules:

- If the data contains all zeros, DAE cannot use it as symptom data. The exception is reason code symptom data. If the SDWARCF flag is on in the SDWA, DAE uses the reason code regardless of its value. RTM turns on the SDWARCF flag whenever the SETRP, CALLRTM, and ABEND macros use the REASON keyword. RTM sets the SDWACRC field equal to the reason code.
- If the data longer than 4 bytes, DAE eliminates leading and trailing zeros.
- If, after eliminating leading and trailing zeros, the symptom is still more than 50 bytes long, DAE truncates the data on the right

Character Data: DAE creates symptoms from character source data according to these rules:

- If the data contains all zeros or all blanks, DAE cannot use it as symptom data.
- DAE eliminates leading and trailing zeros and blanks.
- If, after eliminating leading and trailing zeros and blanks, the symptom is still more than 50 bytes long, DAE truncates the data on the left.

Flag Data: DAE creates symptoms from flag source data according to these rules:

- If the data contains all binary zeros, DAE cannot use it as a symptom.
- If the symptom is more than 50 bytes long, DAE truncates the data on the right.

RETAIN Symptoms

DAE creates RETAIN symptoms according to certain rules. The same source data might yield a RETAIN symptom that differs from an MVS symptom. In creating RETAIN symptoms, DAE first applies its rules for editing MVS symptoms. Then the edited MVS symptom becomes the source for the RETAIN symptom. DAE further edits the data to create RETAIN symptoms according to these rules:

- The format of a symptom is *keyword/data*.
- The length of the *keyword* is variable, and the length of the *data* is variable.
- The maximum length of the symptom is 15 bytes, including the keyword, the slash (/) separator, and the symptom data.
- RETAIN allows use of only three special characters in a symptom; these are:
 - @ (at symbol)
 - \$ (dollar sign)
 - # (pound sign)

DAE replaces all other special characters and blanks with # (pound sign).

In addition, DAE follows certain guidelines for different types of source data.

Hexadecimal and Character Source Data: DAE creates RETAIN symptoms from hexadecimal and character MVS symptom data according to this rule:

- If a symptom is more than 15 bytes long, DAE truncates the data on the left.

Flag Source Data: DAE creates RETAIN symptoms from flag MVS symptom data according to this rule:

- If a symptom is more than 15 bytes long, DAE truncates the data on the right.

Duplicate Symptoms

Sometimes DAE locates duplicate symptoms. For instance, the ABDUMP symptom area contains fields that are also defined in the SDWA. For such duplicate information, DAE uses the ABDUMP data in preference to the same data from the SDWA. If the ABDUMP data areas are blank or contain zeros, DAE uses the data in the SDWA for creating the symptom. Figure 17-10 shows duplicate areas in the ABDUMP symptom area and the SDWA that DAE uses in constructing symptoms.

ABDUMP	SDWA	Description
PRDSMABD	SDWAABCC	System and user abend codes
PRDSMPSW	SDWANXTI	Failing PSW
PRDSMLMN	SDWAMODN	Name of failing load module
PRDSMPDA	SDWAFAIN	Address of failing instruction
PRDSMGPR	SDWAGRSV	General purpose registers at time of failure

Figure 17-10. Duplicate Areas in ABDUMP Symptom Area and SDWA

Multiple Specifications of VRA Keys

You can use the VRADATA macro (see “Creating and Modifying Symptom Data”) to provide specific values for the SDWA VRA keys. If DAE finds multiple specifications for a VRA key, DAE builds the symptom using the first specification of that VRA key.

Criteria for DAE to Match for Duplicates

After DAE creates the MVS and RETAIN symptom strings, it determines if the parameter record specifies or implies **MATCH**. If so, DAE determines if the criteria for matching are present. DAE proceeds with matching for duplicates if the following requirements are met:

- DAE is initialized successfully and no major DAE or SDUMP failures occur during the creation of the MVS symptom string.
- The DAE parameter record specifies or implies matching.
- All required symptoms are nonzero and nonblank. This means that DAE created symptoms and placed them in the MVS symptom string.
- The MVS symptom string meets the minimum length and minimum count requirements.
- The source data areas that DAE uses to search for symptom data are available. See “ABDUMP Symptom Area of the Dump Header Record” and “SDWA” for a list of these symptom data areas.
- The SDWAVRAM flag is on. This indicates that the data in the VRA was created using the VRADATA macro, or was created in the same format as if the VRADATA macro was used. If DAE does not need the data in the VRA for matching, this flag is not necessary.
- The SDWAURAL field accurately reflects the length of the symptom data that you placed in the VRA. If DAE does not need the data in the VRA for matching, this field is not necessary.

If the requirements for matching are not met, DAE allows SDUMP to proceed with dumping. If the requirements for matching are met, DAE attempts to match the newly-created symptom string to the symptom strings in the in-storage symptom queue.

Criteria for DAE to Suppress Dumps

After matching for duplicates, DAE determines if it should proceed with dump suppression. For DAE to suppress dumps, the following criteria must be met:

- All the requirements for matching are met.
- DAE finds a symptom string in the in-storage symptom queue that is a duplicate of the symptom string from the current dump.

- The DAE parameter record specifies **SUPPRESS** for the type of dump (either **SVCDUMP** or **SYSMDUMP**) that DAE is processing.
- The **VRADAE** key is set (see “VRA Keys for DAE”).
- The active **SLIP** commands for the current dump do not specify the **NOSUP**, **SVCD**, or **TRDUMP** options.
- DAE does not encounter terminating errors while processing this dump.

SUPPRESS and UPDATE Processing

When the DAE parameter record specifies **UPDATE** and **SUPPRESS**, DAE does the following processing after match processing:

- If DAE locates a duplicate record:
 1. DAE places the symptom data from the original occurrence of the dump in the dump header record.
 2. DAE updates the in-storage symptom queue record.
 3. If all suppression criteria are met, DAE sets a return code of four; this prevents **SDUMP** from taking the dump.
 4. DAE updates the incidence count of this dump in the corresponding **SYS1.DAE** record.
- If DAE cannot locate a duplicate record in the in-storage symptom queue, it considers the dump to be a new problem.
 1. DAE sets a return code of zero, which allows **SDUMP** to proceed with dumping.
 2. DAE adds a new record to both the in-storage symptom queue and **SYS1.DAE**. The **SYS1.DAE** record contains the symptom string representing this new dump.

The Dump Header Record

DAE provides **SDUMP** and **ABDUMP** with the results of its processing. If DAE does not suppress the dump, the DAE dump header record formatting program takes informational data from the dump header record about why DAE did not suppress the dump. On the dump title page, DAE might list any of the following as reasons for not suppressing the dump:

- The dump is unique; DAE did not find any record of a previous dump.
- **SLIP** requested that the dump be taken.
- **VRADAE** is not set.
- The DAE parameter record does not specify **SUPPRESS** for this dump type.
- DAE could not locate all the required symptoms.
- DAE could not locate the minimum number of symptoms.
- DAE could not build a symptom string of the minimum length.

- DAE encountered a user input error, as follows:
 - Required symptom list has an invalid key.
 - Minimum symptom string count specified is invalid.
 - Minimum symptom string length specified is invalid.

If the dump is a duplicate, DAE places informational data in the dump header record about both the original dump and the current dump.

Some user input errors do not necessarily affect whether or not DAE suppresses dumps. For the following user input errors, DAE still puts information on the dump title page that it takes from the dump header record:

- Optional symptom list has an invalid key.
- SDWAURAL (length of user-supplied information in VRA) or SDWAVRAL (length of VRA) value is invalid.

Overrides to DAE

The following are a few of the reasons why DAE might not suppress dumps:

- If the VRADAE key is not set on, DAE does not suppress that dump.
- The DAE parameter record does not specify the **SUPPRESS** option for this dump type. An installation might omit the **SUPPRESS** option during testing of DAE, but should otherwise specify **SUPPRESS** for improved system performance.
- Certain SLIP actions override DAE dump suppression.
 - SLIP actions of **SVCD** (SVC dump) or **TRDUMP** (trace dump) always produce dumps. These actions trap critical problems that always require a dump.
 - The SLIP action of **NOSUP** (no suppression) always produces a dump.

Creating and Modifying Symptom Data

To use DAE most effectively, you might want to modify the defaults provided in module ADYDFLT, to tailor DAE processing to your installation's needs. Any of the following procedures might provide your installation with better problem determination:

- Adding to the required symptoms
- Adding to the optional symptoms
- Increasing the minimum number of symptoms
- Increasing the minimum symptom string length

When DAE creates a symptom string, it obtains symptoms from three main sources:

- RTM, which creates some symptoms for all dumps except SLIP dumps.
- A program, which creates symptoms when its ESTAE or FRR receives control.
- ABDUMP, which creates symptoms for every SYSMDUMP.

A recovery routine can dynamically define symptom data for DAE to use, by putting specific values into the SDWA variable recording area (VRA).

Two macros allow a recovery routine to access the VRA. The first is the IHAVRA mapping macro, which defines the symptom keys that a recovery routine can put into the VRA. The second is the VRADATA macro, which a recovery routine can issue to put specific data into the SDWAVRA.

This section discusses the VRA keys that DAE uses, and describes how a recovery routine can define additional symptoms for DAE processing. Users must select symptoms carefully. If the symptom data is too specific, no other dump will have the same symptoms, and DAE will not suppress any dumps. If the symptom data is too general, many dumps will have the same symptoms, and DAE might suppress a dump that is needed.

VRA Keys for DAE

An ESTAE or FRR can place information into the SDWAVRA that DAE later uses for creating symptoms. A recovery routine can specify symptom data by using VRA keys, as described below. These keys describe symptom data in the VRA, SDWA, and ABDUMP symptom area that DAE uses for error analysis. The IHAVRA macro defines keys for all symptoms. The VRADATA macro allows the failing component's recovery routine to put data into the SDWAVRA.

If your recovery routine places information into SDWAVRA, it must provide the information in *key/length/data* format. Specifically:

key

The *key* is a one-byte hexadecimal number from X'01' to X'FF' that defines the type of data in the *data* field.

length

The *length* is a one-byte hexadecimal number from X'01' to X'FF' that defines the length of the *data* field.

data

The *data* field contains symptom data determined by the recovery routine.

A recovery routine can issue the VRADATA macro to create a VRA entry in *key/length/data* format. VRADATA also creates the proper length field in the SDWAURAL (length of user-supplied information in VRA).

DAE, as supplied by IBM, defines the meaning and content of the keys in the VRA. However, the keys in the range X'DC' to X'E0' (decimal 220 to 224) are available for whatever you define them to be. You might use these keys when no other keys provide the desired symptoms. You can define a special meaning for one of these keys, and place the key in the SDWAVRA by issuing the VRADATA macro. If you use one of these keys, DAE identifies the corresponding visible keyword as the key preceded by '@'.

The following VRA keys do not represent symptoms, but DAE does use these VRA keys for processing:

VRADAE

enables DAE to suppress duplicate dumps if all requirements for matching and dump suppression are met. To set the VRADAE indicator, the recovery routine issues the following macro instruction:

```
VRADATA KEY(VRADAE)
```

Specification of this key indicates to DAE that the SDWA contains sufficient data to uniquely identify the dump.

Note: Even when the DAE parameter record specifies **SUPPRESS**, if the VRADAE key is not set on, DAE does not suppress any dumps.

VRAMINSC

causes DAE to use the specified value as the required minimum symptom count, if the specified value is not less than the default or more than the maximum.

VRAMINSL

causes DAE to use the specified value as the required minimum symptom string length, if the specified value is not less than the default or more than the maximum.

VRAOPT

identifies a list of VRA keys for this dump that are added to the optional symptom key list specified by the default module ADYDFLT.

VRAREQ

identifies a list of VRA keys for this dump that are added to the required symptom key list specified by the default module ADYDFLT.

Note: You can use the VRADATA macro to provide specific data for the SDWAVRA. If DAE finds multiple specifications for a VRA key, DAE builds the symptom using the first specification of that VRA key.

Adding to the Minimum Symptom String Requirements

The ADYDFLT module contains defaults for the required symptoms, optional symptoms, minimum number of symptoms, and minimum symptom string length for DAE processing. (See "ADYDFLT" in this chapter for a discussion of the defaults it contains.) You can use appropriate data areas in the SDWA to modify these minimum requirements.

Adding to the Required Symptoms

You can dynamically add to the required symptoms that DAE uses for matching. Use the VRADATA macro to create a field in SDWAVRA in *key/length/data* format, using the VRAREQ key. The *data* field must contain binary numbers corresponding to the labels in the IHAVRA mapping macro. The *length* must be a multiple of two.

DAE adds these keys to the default set specified by module ADYDFLT. The maximum number of required symptoms that DAE can use is 20. If you add more than 18 required symptoms to the two provided by ADYDFLT, DAE uses the first 20 and ignores the rest.

If you create the VRAREQ field more than once, DAE accumulates the specified required symptoms from the data portion of each field until it reaches the maximum of 20.

Notes:

- 1. If information is missing from a required symptom (that is, a symptom is all zeros or all blanks), DAE does not match for duplicates. The title page of the dump lists a DAE informational message to this effect.*
- 2. If you include an invalid key for a required symptom, DAE does not match for duplicates. The title page of the dump lists a DAE informational message about the invalid key.*

Adding to the Optional Symptoms

You can dynamically change the number and type of optional symptoms that DAE uses for matching. Use the VRADATA macro to create a field in SDWAVRA in *key/length/data* format, using the VRAOPT key. The *data* field must contain binary numbers corresponding to the labels in the IHAVRA mapping macro. The *length* must be a multiple of two.

DAE adds these symptom keys to the default set defined by module ADYDFLT. The maximum number of optional symptoms that DAE can use is 20. If you add more than 12 optional symptoms to the eight provided by ADYDFLT, DAE uses the first 20 and ignores the rest.

If you create the VRAOPT field more than once, DAE accumulates the specified optional symptoms until it reaches the maximum of 20.

Note: If you include an invalid *key* for an optional symptom, DAE ignores this key, but still matches for duplicates. The title page of the dump lists a DAE informational message about the invalid key.

Increasing the Minimum Number of Symptoms

You can increase the minimum number of symptoms that DAE uses for matching. Use the VRADATA macro to create a field in SDWAVRA in *key/length/data* format, using the VRAMINSC key. The *data* field is a hexadecimal number from X'05' to X'28' (decimal five to 40). This value defines the number of required and optional symptoms that DAE must find before matching occurs. DAE can use up to 20 required symptoms and 20 optional symptoms for matching, as long as the symptom string does not exceed 150 bytes. The *length* must always equal two.

If you create the VRAMINSC field more than once, DAE uses the count from the last valid VRAMINSC field specified. DAE does not match for duplicates when:

- You specify a value that is less than the default value defined in ADYDFLT.
- You specify a value that is more than 40.

The title page of the dump lists a DAE informational message to this effect.

Increasing the Minimum Symptom String Length

You can increase the minimum symptom string length that DAE requires for matching. Use the VRADATA macro to create a field in SDWAVRA in *key/length/data* format, using the VRAMINSL key. The *data* field is a hexadecimal number from X'19' to X'96' (decimal 25 to 150). This value defines the minimum symptom string length that DAE must use before matching occurs. The *length* must always equal two.

If the VRAMINSL field is created more than once, DAE uses the count from the last valid VRAMINSL field specified. DAE does not match for duplicates when:

- You specify a value that is less than the default value defined in ADYDFLT.
- You specify a value that is more than 150.

The title page of the dump lists a DAE informational message to this effect.

Appendix A. IBM Provided Device Preference Table

The following IBM provided list shows the device order that MVS uses when it attempts an allocation to satisfy a request for a device from an esoteric device group. The order of the IBM-defined list ensures that MVS always tries to allocate the fastest possible available device.

For each UIM that you write, you may add the generic name and generic preference value to this default list by inserting an unsupported device anywhere in the list. While you may add to the list, you can not change the order of the IBM-defined list this way.

For each EDT that you define, you may change the order of the list or add unsupported devices by coding the changes on the `DEVPREF =` parameter of the EDT statement.

Device Type	Generic Name	Generic Preference Value
drum	2305-2	200
direct access	3380 3350 3375 3330-1 3330 3340	290 300 350 400 500 600
magnetic tape	3480 3400-9 3400-5 3400-3 3400-6 3400-4 3400-2 2400-3 2400 2400-4 2400-2 2400-1	1100 1101 1200 1210 1220 1230 1240 1300 1310 1320 1300 1340
printers	3800 4248 ¹ 4245 3211 3203 1403	1780 1850 1890 1900 2000 2100
readers/punches	2501 3505 3525 2540 2540-2	2300 2400 2500 2800 2900
graphic (display) devices	HFGD 2250-3 3277-1 3277-2 3284-1 3284-2 3286-1 3286-2 2260-1 2260-2	3260 3500 3700 3800 4100 4200 4300 4400 4600 4700
optical or magnetic character readers	3890 3886 1287 1288 3895	4800 4900 5000 5100 5400
diskette	3540	5600
mass storage subsystem	3851 3330V	5700 5800
array processor	3838	5900

¹Because the 3262 Model 5 is generated as a 4248 printer, the entry 4248 in the installation device preference table may refer to a 4248 printer, a 3262 Model 5 printer, or both. If a 3262 Model 5 printer is installed on your system, and the 4248 entry in your installation device preference table precedes the entries for the 4245 or the 3800 printers, your system may allocate the slower 3262 Model 5 printer in preference to the faster 4245, 4248, or 3800.

Device Type	Generic Name	Generic Preference Value
telecommunication device	AAA1	6100
	AAA2	6200
	AAA5	6500
	AAA6	6600
	AAA7	6700
	AAA8	6800
	AAA9	6900
	AAAA	7000
	AAAB	7100
	AAAC	7200
	AAAD	7300
	AAAE	7400
	AAAF	7500
	3705	7600
	3791L	7700
3704	7800	
channel-to-channel CU	CTC	8400
telecommunication devices	AAAG	9700
	AAAH	9800
	AAAI	9900
miscellaneous devices	DUMMY	99991
	3848	99999

Index

A

ABDUMP symptom area of dump header record 17-34
activating general-purpose IEAVMXIT 7-6
adding an installation-written resource manager 5-4
adding entries to the system trace table
 See PTRACE macro
adding to minimum symptom string for DAE 17-42
adding to optional symptoms for DAE 17-43
adding to required symptoms for DAE 17-43
adding your own code to the DAT-off nucleus 6-1
address space for allocation/deallocation
 See ALLOCAS address space
ADSSRNSD data area and DAE 17-34
ADYDFLT module (DAE)
 adding to minimum symptom string 17-42
 description 17-30
 symptom keys
 adding to defaults 17-31
 optional 17-31
 required 17-31
affinity, processor 4-3
ALLOCAS address space 2-13
allocating SYS1.DUMP data sets 17-5
allocating the internal reader data set 14-2
allocation
 See also device preference table, volume attribute list, mount and use attributes
 deallocation of resources in case of abend 2-13
 general discussion 2-1
 improving efficiency of 2-2
 order of device selection 2-1, 2-2
 serialization during allocation 2-1
allocation considerations 2-1
AMDDATA mapping macro 17-34
AMDPRDMP
 Defining ABEND Dump Data Sets 17-3
 SYSMDUMP 17-3
 Formatting SVC dumps 17-4
 Printing SVC dumps 17-4
AMODE and RMODE attributes, in DAT-off code 6-1
APF-authorization
 required for programs in the PPT 4-5, 4-6
assembler code for creating USRn TTEs 9-7
assigning mount and use attributes 2-8
assigning special program properties to applications 4-1
 See also program properties table
automatic dump suppression
 See DAE
 See suppressing dumps automatically, by abend code

B

block character routine for output separator pages
 See output separator routine for an external writer
build function of virtual fetch 16-6
 See also virtual fetch
BUILD request for virtual fetch 16-13

C

CBPADIT 1-5
CBPDDCT 1-6
CBPIDFT 1-7
CBPIFEAT 1-8
CBPIGETM 1-9
CBPIPARM 1-10
CBPZDIAG 1-38
CBPZLOG 1-38
CBPZPPDS 1-38
changes to Master JCL 8-2
character source data for DAE
 MVS symptoms 17-36
 RETAIN symptoms 17-37
check groups function of unit verification service 12-1
check units function of unit verification service 12-2
CHNGDUMP command 17-3
closing the internal reader data set 14-3
coding a formatting routine for USRn TTEs
 See writing an ITRF0n7F TTE formatting routine
coding conventions for the output writing routine 15-4
commands
 See system operator commands
Configuration program vector table 1-3
considerations when using virtual fetch 16-9
console communications, controlling
 See WTO/WTOR exits
contents of the PPT entry 4-3
continued USRn trace table entries
 See multi-part trace table entries
Control blocks 1-3
control points for customizing the system iii
 job/step execution iii
 system initialization (IPL) iii
controlling GRS requests in MVS/XA 2-13
controlling system messages 7-2
 processing 7-3
 routing 7-3
controlling system messages and the system log 7-1
controlling the system log 7-8
convert device type to look-up value function of unit verification service 12-2
 output return codes 12-15
CPU affinity 4-3

CPVT 1-3
 creating your own resource managers 5-1
 CSVVFCRE 16-2, 16-7
 See also virtual fetch
 CSVVFGET 16-6
 CSVVFGTE 16-6
 CSVVFFIND 16-12
 CSVVFNDE 16-6
 CSVVFSCH 16-2, 16-6
 CSVVFTCH 16-6
 customization task, the iii
 customizing the system trace table 9-1
 sample assembler code for 9-7
 customizing your system
 installation defaults iv
 macro instruction libraries iv
 module libraries iv
 system operator commands iv
 system services iv
 SYS1.PARMLIB iv
 user exits iv
 CVTVFGET 16-6
 See also virtual fetch
 CVTVFFIND 16-6
 See also virtual fetch

D

DAE
 adding to
 minimum symptom string 17-42
 optional symptoms 17-43
 required symptoms 17-43
 changing symptom keys in ADYDFLT 17-31
 creating symptom data 17-35, 17-40
 default keys in ADYDFLT 17-30
 dump header record 17-39
 functions 17-26
 increasing
 minimum number of symptoms 17-44
 minimum symptom string length 17-44
 initialization 17-35
 input data sources 17-29
 modifying symptom data 17-40
 overrides
 SLIP actions 17-40
 VRADAE key 17-40
 overview 17-26
 parameter record
 description 17-29
 format 17-30
 processing
 summary 17-34
 SUPPRESS 17-26, 17-39
 UPDATE 17-26, 17-39
 requirements
 for matching 17-38
 for suppressing 17-38

 symptom data
 extraction 17-35
 searching for 17-32
 VRA keys in SDWA 17-38, 17-41
 DAEALLOC member of SYS1.SAMPLIB 17-32
 DAT-off code in MVS/XA 6-1
 Data Areas
 data set processing subtask
 See STDWTR: IBM standard output writing routine
 DATOFF macro 6-1
 DCT 1-6
 DDT 1-38
 device descriptor table 1-33
 deallocation of resources in case of abend 2-13
 using the ALLOCAS address space 2-13
 default values in IOSRHIDT 13-2
 defining DUMMY devices 1-1
 defining dump data sets
 for ABEND dumps 17-3
 for SVC dumps 17-5
 defining subsystems in members of
 SYS1.PARMLIB 10-2
 entry format 10-2
 in IEFSSNxx 10-2
 defining subsystems to the operating system 10-1
 at system generation 10-1
 in members of SYS1.PARMLIB 10-2
 initialization routines for subsystems 10-1
 parameters for initialization routines 10-3
 system handling of duplicates 10-4
 defining the job entry subsystem 8-1
 descriptor codes 7-3
 See also WTO/WTOR exits
 Device Characteristics Table
 CBPDDCT 1-6
 Device class
 unit names for 12-3
 Device dependent table
 for UIMs 1-38
 IOSDDT macro 1-38
 Device descriptor table
 IOSDDT macro 1-33
 Device Features Checker
 CBPIFEAT 1-8
 Device Features Table
 CBPIDFT 1-7
 Device Information Table
 CBPADIT 1-5
 device preference table 2-2, 2-3
 defined in the EDT macro 2-3
 IBM-defined values A-1
 device support modules
 in SYS1.LINKLIB 1-33
 see also DDT 1-33
 in SYS1.NUCLEUS 1-33
 module list table 1-33
 see also MLT 1-33
 device verification 12-1
 devices
 defining DUMMY device 1-41, 1-42

- defining non-supported devices 1-1
- DUMMY 1-1
- I/O configuration 1-1
- IBM-supplied UIMs 1-39
- MVSCP 1-1
- to support MVSCP 1-39
- Unit Information Module 1-1
- DEVPREF parameter of the EDT macro 2-3
- DFT 1-7
- Diagnostics for UIMs
 - Adding entries to the diagnostic stack 1-38
 - CBPZPPDS 1-38
 - Messages 1-38
 - CBPZLOG macro 1-38
 - message identifiers 1-38
 - MVSCP's recovery routine 1-38
 - CBPVMVSCP 1-38
 - CBPZDIAG macro 1-38
 - taking entries off the diagnostic stack 1-38
 - CBPZPPDS 1-38
- DISPLAY DUMP command
 - See system operator commands
- DIT 1-5
- DUMMY device, coding for a 1-42
- dump analysis and elimination
 - See DAE
- DUMP command 17-7
- dump header record 17-39
- dump suppression, function of DAE 17-26
- dump types, in MVS 17-2
 - ABEND (user) 17-2
 - JES3 user dumps 17-2
 - SNAP 17-2
 - STAND-ALONE 17-2
 - SVC (system) 17-2
- DUMPDS command 17-20
- dumping services
 - See MVS dumping services
- Dumps
 - Machine readable 17-25
 - BLSCECT parmlib member 17-25
 - IPCS dialogs 17-25
 - IPCSPRxx parmlib member 17-25
- duplicate subsystem names 10-4
- duplicate symptoms for DAE 17-37
- dynamic allocation 2-1

E

- EDT
 - See eligible device table
- eligible device table 2-2, 2-6
 - creating new tables using EDT statement 2-7
 - Unit verification service 12-1
 - IEFAB4UV 12-1
 - IEFEB4UV 12-1
 - IEFGB4UV 12-1
- ERPINDEX 1-42

- esoteric group names 2-6
 - example of setting up 2-6
 - figure showing relationships to generics 2-7
 - specified on the UNITNAME statement 2-6
- ESTAE
 - See recovery routines
- examples of using preferred storage flags 4-8
- executing DAT-off code in MVS/XA 6-1
- extended addressing considerations 3-2, 6-1, 14-1, 15-1, 16-11
- the external writer 15-1
 - See also external writer
 - block characters to separate jobs 15-11
 - default processing 15-1
 - defined 15-1
 - features of IBM-supplied version 15-1
 - initialized by module IASXWR00 15-2
 - modifying the IBM version 15-1
 - output separation processing 15-8
 - parameter list 15-2
 - format and contents 15-3
 - parts of 15-1
 - output separator routine 15-1, 15-8
 - output writing routine 15-1, 15-2
 - selection criteria for data sets 15-1
 - IBM default 15-1
 - using the MODIFY command 15-13
 - using the 3800 Printing Subsystem 15-15
 - XWTR, IBM cataloged procedure 15-12
- the external writer cataloged procedure 15-12, 15-14
 - for the 3800 Printing Subsystem 15-15
 - modifying for special print chains 15-15
- external writer parameter list (PARLIST) 15-2

F

- find function of virtual fetch 16-6
 - See also virtual fetch
- FIND request for virtual fetch 16-13
- fitting your subsystems into the system 9-11
- fixed pages
 - See preferred storage flags in the PPT
- flag source data for DAE
 - MVS symptoms 17-36
 - RETAIN symptoms 17-37
- format of the PPT entry 4-2
- formatting a USRn trace table entry 9-4
 - handling errors 9-7
 - parameters passed to print routine 9-6
 - parameters received 9-5
 - possible return codes 9-6
 - writing your own routine 9-5-9-7
- formatting USRn trace table entries
 - See writing an ITRF0n7F TTE formatting routine
- FRR
 - See recovery routines
- function codes for unit verification service 12-5

functional parts of external writers
See external writer
functions of virtual fetch 16-1

G

general-purpose WTO exit
capabilities 7-5
IEAMVXIT 7-4
routine's function 7-4
generic and esoteric device groups 2-7
generic device types 2-3
on the device preference list 2-3
relationship to esoteric group names 2-7
get function of virtual fetch 16-6
See also virtual fetch
GET request for virtual fetch 16-14
GETMAIN limits set by VSM 3-2
effect of region size and limit values on GETMAIN requests 3-9
example using the IEFUSI interface 3-7
examples of allocations using values from IEALIMIT 3-10
function of the GETMAIN limit 3-9
using values from IEALIMIT 3-3
defaulting to no limit 3-3
using values from IEFUSI 3-2
GETMAIN requests 3-3
fixed-length 3-9
examples of requests 3-9, 3-10
variable-length 3-3, 3-9
examples of requests 3-9, 3-10
how to avoid using up private area 3-10
Getmain Service Routine
CBPIGETM 1-9
GNRCPT1 1-42
GNRCTYP1 1-42
GRNCNM 1-42
GRS requests, controlling the number of 2-13

H

handling errors during TTE formatting 9-7
hardcopy log 7-7
activation 7-7
commands on 7-8
messages deleted from 7-7
messages on 7-7
on the system log data set 7-7
hash table for virtual fetch
See virtual fetch
hexadecimal source data for DAE
MVS symptoms 17-36
RETAIN symptoms 17-37
HIDT

See hot I/O detection table
hot I/O condition 13-1
definition 13-1
detected by IOS 13-1
hot I/O detection table 13-1
device threshold value in 13-1
IBM default values in 13-2
mapped by macro IOSDHIDT 13-1
modifying 13-3
recovery action codes in 13-1
how this book is organized v
how VSM uses the region size and GETMAIN limits 3-9

I

I/O configurations 1-1
IASXWR00
See external writer
IBM-supplied UIMs 1-39
IEACMD00 member of SYS1.PARMLIB
automatic start-up of DAE 17-29, 17-35
description 17-30
IEALIMIT exit 3-2
processing by IBM version 3-3
replacing IBM's routine 3-2
requesting VSM defaults 3-4
result of setting no limits in 3-3
setting user region limits in MVS/370 3-2
IEALIMIT processing 3-3
IEASYSxx members of SYS1.PARMLIB
See SYS1.PARMLIB
IEAVEDAT in SYS1.NUCLEUS 6-1
IEAVETEF, system trace filter module 9-4, 9-5
IEAVETFC, load module for USRn TTE
formatter 9-5
IEAVETPB, print routine for the system trace output
buffer 9-6
IEAVEURn entry point for DAT-off code 6-1
IEAVMXIT 7-7
See also general-purpose WTO exit
procedure for
user-specified exit 7-7
IEAVMXIT - general-purpose WTO exit 7-3
and user-specified WTO exits 7-4
capabilities 7-3
functions 7-4
use in suppressing messages 7-8
IEAVTRML
See resource managers, adding your own
IECIOSxx members of SYS1.PARMLIB
See SYS1.PARMLIB
IEEMSJCL 8-1
IEFAB4UV 12-1, 12-4
addressing mode 12-4
authorized callers 12-4
caller's function 12-4

- key 12-4
- mode 12-4
- IEFEB4UV 12-1, 12-4
 - See also unit verification service
 - authorized callers 12-4
 - caller's function 12-4
 - key 12-4
 - mode 12-4
 - problem program callers 12-4
- IEFGB4UV 12-1, 12-4
 - authorized callers 12-4
 - caller's function 12-4
 - key 12-4
 - mode 12-4
- IEFGB4UV or IEFAB4UV 12-1
 - See also unit verification service
- IEFJESCT mapping macro 11-4
- IEFJSSIB 11-5
- IEFSDPPT 4-1
 - See also program properties table
 - IBM-supplied PPT 4-4
 - overriding PPT entries 4-1
 - SCHEDxx member of SYS1.PARMLIB 4-1
- IEFSD087
 - See STDWTR: IBM standard output writing routine
- IEFSD094: the output separator routine 15-8
- IEFSD095
 - See output separator routine
- IEFSSNxx parmlib members 10-2
 - See also defining subsystems in members of SYS1.PARMLIB
 - entry format 10-2
 - IEFSSN00, IBM default 10-2
 - including parameters for the initialization routine 10-2
 - specified on SSN = system parameter 10-2
- IEFSSOBH mapping macro 11-4
- IEFSSREQ macro
 - See subsystem affinity service
- IEFSSREQ: obtaining the SSAT index value 11-4
- IEFSSVS 11-5
- IEFUSI exit 3-4
 - example of IEFUSI processing 3-8
 - requesting VSM defaults 3-4
 - setting user region limits in MVS/XA 3-2
- IEFUSI interface 3-4
- IEFZB610 4-1
 - See also program properties table
- IHAVRA mapping macro 17-41
- improving allocation performance 2-2
- increasing minimum number of symptoms for DAE 17-44
- increasing minimum symptom string length for DAE 17-44
- initialization routines for subsystems 10-1
 - parameter list 10-3
 - format 10-3
- mapped by IEFJSIPL, in SYS1.MACLIB 10-3
- input to and output from unit verification service routines 12-5
- inserting a WTO/WTOR exit routine into the control program 7-5
- installation support for the virtual fetch service 16-6
- installation-written resource managers 5-1
- the internal reader facility 14-1
 - allocating the data set 14-2
 - notes 14-2
 - via dynamic allocation (SVC 99) 14-2
 - via JCL 14-2
 - closing the data set 14-3
 - coding /*DEL 14-4
 - coding /*EOF 14-3
 - coding /*PURGE 14-4
 - using the CLOSE macro 14-4
 - when the buffer is full 14-3
 - definition 14-1
 - example 14-4
 - opening the data set 14-3
 - sending records to the data set 14-3
 - starting a secondary subsystem via an internal reader 14-6
 - tasks involved in using 14-1
- IODEVICE
 - Internal text record 1-3
 - IODEVICE internal text record 1-3
 - IODEVICE Parameter Checker
 - CBPIPARM 1-10
 - IODEVICE statement 1-1
 - limits of DUMMY definition 1-1
 - Processing 1-2
- IODV 1-3
- IOSDDT 1-38
- IOSDDT macro 1-38
- IOSDDT macro instruction
 - syntax 1-35
- IOSDHIDT 13-1
 - See also hot I/O detection table
- IOSDMLT 1-38
- IOSDMLT macro 1-38
- IOSDMLT macro instruction
 - syntax 1-37
- IOSRHIDT: the HIDT 13-1
- IPCS
 - Defining ABEND Dump Data Sets 17-3
 - SYSMDUMP 17-3
 - Formatting SVC dumps 17-4
 - Printing SVC dumps 17-4
 - for SVC dumps 17-4
- issuing the START command to a secondary subsystem 14-6
- ITRFDEFU, default formatter of trace table entries 9-5
- ITRF007F, or ITRF0n7F 9-4

J

- JCL REGION parameter
 - See REGION parameter on the JOB/EXEC statement
- JCL, master 8-1
- JES
 - See also job entry subsystem
 - defining 8-1
 - IEFSSNxx 8-1
 - Master JCL 8-1
 - START command 8-1
- JES2 4-8, 14-3
 - See also job entry subsystem
 - as example of PPT entry 4-8
 - control statements for internal reader 14-3, 14-4
- JES3 14-3
 - See also job entry subsystem
 - action messages unsuppressible 7-8
 - format of WTL macro output 7-9
 - handling internal reader data 14-3
- JES3 user dumps
 - See dump types, in MVS, ABEND (user)
- job entry subsystem 7-9, 7-10, 16-2
 - and the external writer 15-1, 15-2, 15-12
 - and the internal reader 14-1, 14-2, 14-3
 - defining to the operating system 9-11
 - See also primary subsystem, defining
 - differences in hard copy and system logs in 7-7
 - REGION default for IEALIMIT 3-3
 - starting a secondary subsystem 14-7
 - system message text modified by 7-3
- job entry subsystem, defining 8-1

K

- keys for DAE
 - definition 17-27
 - in ADYDFLT
 - optional 17-31
 - required 17-31
- keywords
 - definition for DAE 17-27
 - MVS 17-28
 - RETAIN 17-28
- kinds of dumps produced in MVS
 - See dump types, in MVS

L

- libraries for system modules
 - See system module libraries
- limiting user region size 3-1
 - by virtual storage management 3-1
 - figure showing default limits set by VSM 3-5
 - for V = R jobs 3-11
 - how VSM uses region size and GETMAIN
 - limits 3-9
 - setting JCL defaults 3-1
 - setting values in exit routines 3-1
 - IEALIMIT, for MVS/370 3-2
 - IEFUSI, for MVS/XA 3-2
- Link library
 - &I2@lnklib.
 - for resource manager routines 5-4
- LOGCLS system parameter 7-10
- LOGLMT system parameter 7-10
- look-up value for the EDT
 - See also unit verification service
 - defined 12-2
 - obtaining 12-2
- LPALIB
 - See system module libraries

M

- Machine Readable Dumps
 - BLSCECT parmlib member 17-25
 - IPCPSRxx parmlib member 17-25
 - IPCS dialogs 17-25
- Macros
 - required in UIMs 1-38
 - CBPZDIAG 1-38
 - CBPZLOG 1-38
 - CBPZLOGR 1-38
 - CBPZPPDS 1-38
 - IOSDDT 1-38
 - IOSDMLT 1-38
- managing modules
 - See virtual fetch
- Master JCL
 - adding 8-2
 - alternate versions of 8-2
 - changing 8-2,
 - AMASPZAP 8-2
 - defining the job entry subsystem 8-1, 8-2
 - deleting 8-2
 - IBM-supplied sample 8-2
 - SAMP MJCL 8-2
 - SYS1.SAMPLIB 8-2
 - IEEM SJCL 8-1
 - example 8-1
 - IEFSSNxx 8-2

- modifying 8-2
 - adding DD statements 8-2
 - deleting the JES START command 8-2
- modifying SAMPJCL 8-2
- MSTJCL00 8-1
- notes on 8-2
- START command 8-1
- Updating 8-1
- Master Subsystem
 - JCL restrictions with START SUB=MSTR 14-6
 - restrictions with a started task 14-6
- matching for duplicate dumps (DAE) 17-26
- MCS
 - See multiple-console support
- Message ids for UIMs 1-15
- message processing facility 7-1
 - activating 7-8
 - suppressing message display 7-8
- message routing codes
 - See also WTO/WTOR exits
 - functions 7-2, 7-3
 - in hardcopy log 7-7
- messages
 - controlling the routing of 7-1, 7-2
 - in the hardcopy log 7-7
 - suppressing display of 7-8
- minimum symptom string for DAE
 - adding to 17-42
 - increasing length of 17-44
- minimum symptoms for DAE
 - description 17-28
 - increasing number of 17-44
- MLT 1-38
 - in SYS1.LINKLIB 1-33
 - in SYS1.NUCLEUS 1-33
 - IOSDMLT macro 1-33
- MODIFY command
 - See external writer
- modifying the HIDT 13-3
- modifying the operating system: an overview iii
 - protecting system integrity iii
- modifying the system log 7-9
- modifying the system to fit the devices in your installation 1
- modifying the system to fit your applications 2-15
- module libraries
 - See system module libraries
- module list table
 - for UIMs 1-38
 - IOSDMLT macro 1-38
- mount and use attributes for volumes 2-7
 - assigning 2-7
 - using a VATLSTxx parmlib member 2-7
 - using the MOUNT command 2-7
 - definitions 2-7
 - mount attribute of permanently resident 2-8
 - mount attribute of removable 2-8
 - mount attribute of reserved 2-8
 - use attribute of private 2-7

- use attribute of public 2-7
 - use attribute of storage 2-8
- relationships to allocation requests 2-10
- the non-sharable attribute 2-10
- MPF
 - See message processing facility
- MPFLSTxx parmlib members
 - See SYS1.PARMLIB
- MSTJCL00 8-1
- multi-part trace table entries 9-3
 - See also PTRACE macro
- multiple specifications of VRA keys (DAE) 17-38
- multiple-console support 7-1
 - routing messages 7-2
- MVS configuration program 1-1
- MVS dumping services 17-1
 - kinds of dumps produced 17-2
 - tasks accomplished with 17-1
 - defining dump data sets 17-3, 17-5
- MVS dumps
 - See dump types, in MVS
- MVS/XA system services 11-7
- MVSCP
 - calls to the UIM 1-11
 - end-of-processing 1-11
 - initialization call 1-11
 - IODEVICE statement checking 1-11
 - defining devices 1-1
 - logic of processing 1-11
 - recovery routine 1-38
 - CBPVMVSCP 1-38
 - vector table 1-3
- MVSCP Service Routines
 - CBPADIT 1-5
 - Builds DITs 1-5
 - CBPDDCT 1-6
 - Device Characteristics Table 1-6
 - CBPIDFT 1-7
 - CBPIFEAT 1-8
 - CBPIGETM 1-9
 - CBPIPARAM 1-10

N

- NAMEMLT 1-42
- new addressing mode
 - See extended addressing considerations
- no validity bit function of unit verification service 12-3
- non-JES output writing routine
 - See external writer
- non-sharable attribute 2-10
 - relationship between non-sharable and sharable
 - allocation requests 2-11
- notes on the preferred storage flags 4-5
- notes on updating the PPT 4-9
- notes on using the program properties flags 4-5

O

- OPEN macro
 - J-type issued by STDWTR 15-2
 - opening the input data set 15-3
- opening the internal reader data set 14-3
- operator commands
 - See system operator commands
- optional symptom keys for DAE
 - in ADYDFLT 17-31
- optional symptoms for DAE
 - adding to 17-43
 - definition 17-29
- order of allocation A-1
- other MVS/XA publications vii
- out-of-space condition 3-3
 - See also virtual storage management, limiting user region size
- output from the separator routine 15-10
- output separator routine for an external writer 15-8
 - block character routine supplied by IBM 15-11
 - functions 15-11
 - invoking 15-11
 - module name IEFSD095 15-11
 - parameter list 15-11
 - functions of IBM version 15-8-15-9
 - module name IEFSD094 15-8
 - note on protection key 15-11
 - output from the routine 15-10
 - parameter list 15-9
 - requirements for writing your own 15-10
 - return code 15-11
- output writer
 - See external writer
- output writing routine for an external writer 15-2
 - See also external writer
 - coding conventions 15-4
 - creating your own 15-3
 - handling errors 15-8
 - obtaining storage for work areas 15-6
 - processing the input data set 15-6
 - closing the input data set 15-8
 - EOF on input data set 15-7
 - handling record control characters 15-6
 - initializing the printer 15-6
 - reading the input records, using GET macro 15-6
 - reconciling input with output data set 15-7
 - releasing storage 15-8
 - setting return codes 15-4, 15-8

P

- parameter record for DAE 17-29
- parameters and parameter lists
 - See also individual topic headings
 - for a subsystem's initialization routine 10-1, 10-3
 - for ABEND and SVC dumps 17-7
 - for post-dump exit routines 17-23
 - for pre-dump exit routines 17-22
 - for the external writer (PARLIST) 15-2
 - for the USRn formatting routine 9-5
 - for virtual fetch service 16-6, 16-9, 16-10, 16-11
 - from RTM for resource managers 5-2
 - from SMF for IEFUSI exit routine 3-6
 - in ADYSETxx for DAE 17-26
 - on the SSAFF macro 11-3
 - SUB= parameter on the START command 14-7
 - to control system log activity 7-10
 - to put user data into the system trace table 9-2
- PARMLIB
 - See SYS1.PARMLIB
- passing parameters to a subsystem initialization routine 10-3
- permanently resident volume 2-9
 - See also mount and use attributes for volumes, definitions
 - assigning use attributes 2-9
 - notes 2-9
 - volumes that are always permanently resident 2-9
- PPT 4-1
 - See also program properties table
 - APF-authorized 4-5
 - assigning special properties 4-1
 - CSECT IEFSDPPT 4-1
 - entry 4-2, 4-3
 - contents of 4-3
 - format 4-2
 - PPTBYTE1 4-2
 - PPTCPUA 4-2
 - PPTKEY 4-2
 - PPTNAME 4-2
 - PPTORIG 4-2
 - PPTPUBYT 4-2
 - format 4-2
 - PPTENTLN 4-2
 - PPTENTS 4-2
 - PPTHDRLN 4-2
 - PPTIB650 4-2
 - PPTID 4-2
 - PPTMSGAD 4-2
 - PPTUSED 4-2
 - PPTVERS 4-2
 - header 4-2
 - IBM-supplied PPT 4-4
 - IEFSDPPT 4-4
 - macro IEFZB610 4-1
 - notes on using PPTBYTE1 4-5
 - overriding PPT entries 4-2

- PPT entry origin 4-4
 - PPTORIG 4-4
- PPTDEFAULT 4-4
 - residence 4-5
- SCHEDxx member of SYS1.PARMLIB 4-1
- TCAM message control program 4-9
 - using program properties flags 4-5
- V = R 4-4
 - vector programs 4-3
 - with the VARY STOR,OFFLINE 4-4
- PPT entries
 - See program properties table, entries
- PPT entry, contents of 4-3
- PPTBYTE1 4-2
 - See also program properties flags in the PPT
- PPTCPUA 4-2
 - See also processor affinity mask in the PPT
- PPTENTLN 4-2
- PPTENTS 4-2
- PPTHURLN 4-2
- PPTIB650 4-2
- PPTID 4-2
- PPTKEY 4-2
 - See also protection key in the PPT
- PPTMSGAD 4-2
- PPTNAME 4-2
 - See also program name in the PPT
- PPTORIG 4-2, 4-4
- PPTPUBYT 4-2
 - See also preferred storage flags in the PPT
- PPTUSED 4-2
- PPTVERS 4-2
- PRDMP
 - Defining ABEND Dump Data Sets 17-3
 - SYSMDUMP 17-3
 - Formatting SVC dumps 17-4
 - Printing SVC dumps 17-4
- PRDSYSMD data area and DAE 17-34
- pre-dump exits for user dumps 17-22
- Preference table values
 - IBM-supplied A-1
- preferred storage flags in the PPT 4-3
 - common uses of 4-9
 - examples of using 4-8
 - for users of SYSEVENT TRANSWAP 4-5
 - notes 4-5
 - for swappable programs 4-5
 - relationship to ASCB flags 4-5
 - PPTN2LP: no short-term pages to preferred frames 4-4
 - PPT1LPU: long-term pages to preferred frames 4-4
 - PPT2LPU: short-term pages to preferred frames 4-4
 - tips on using 4-5, 4-6-4-7
- primary subsystem, defining 10-1
- printing the trace output buffer contents 9-6
- private volume 2-8
 - See also mount and use attributes for volumes, definitions
- Processing Machine Readable Dumps
 - BLSCECT parmlib member 17-25
 - IPCPSRxx parmlib member 17-25
 - IPCS dialogs 17-25
 - processor affinity mask in the PPT 4-3
 - program name in the PPT 4-3
 - program properties flags in the PPT 4-3
 - notes on using 4-5
 - APF-authorization required 4-5
 - bypass-password-protection property 4-5
 - data set integrity 4-5
 - PPTNCNCL: make program noncancellable 4-3
 - PPTNSDI: data set integrity not required 4-3
 - PPTNOPAS: bypass password protection 4-3
 - PPTNSWP: make program nonswappable 4-3
 - PPTPRIV: make program privileged 4-3
 - PPTSKEY: assign protection key 4-3
 - PPTSYSTK: program is a system task 4-3
- Program Properties Table 4-1
 - assigning special properties to programs 4-1
 - entries 4-1
 - contents 4-3-4-4
 - format 4-2
 - mapping macro IEFZB610 4-1
 - SCHEDxx member of SYS1.PARMLIB 4-1
 - header 4-1
 - meaning 4-1
 - updating 4-9
 - notes 4-9
 - programming conventions for using virtual fetch 16-11
 - protection key in the PPT 4-3
 - protection of SYS1.DAE data set 17-33
- PTRACE macro 9-2
 - continuation information for multi-part TTEs 9-3
 - data to be traced 9-2
 - for system tracing, only 9-2
 - processing 9-3
 - syntax 9-2
- public volume 2-8
 - See also mount and use attributes for volumes, definitions
- Push/Pop macro 1-15

R

- recovery of allocated resources 2-13
- Recovery routine
 - for UIMs 1-38
 - MVSCP's 1-38
 - CBPVMVSCP 1-38
- recovery routines
 - creating symptoms for DAE 17-40
 - modifying symptoms for DAE 17-40
 - use of VRA keys for DAE 17-41
- references for new users of MVS/XA viii
- refreshing virtual fetch 16-6, 16-8
- REGION parameter on the JOB/EXEC statement 3-1, 3-3
 - default if parameter is not specified 3-3

- example using IEFUSI 3-7
- examples of allocations using REGION value 3-10
- limiting V = R jobs 3-11
- setting region size value 3-1
- value if parameter is nonzero 3-3
- removable volume 2-9, 2-10
 - See also mount and use attributes for volumes, definitions
 - assigning use attributes 2-9
- Replacing a WTO/WTOR exit routine without a re-IPL procedure for 7-7
- replacing the standard output writing routine 15-3
- requesting dumps when using virtual fetch 16-11
- required symptom keys for DAE
 - in ADYDFLT 17-31
- required symptoms for DAE
 - adding to 17-43
 - definition 17-29
- reserved attribute 2-9
 - See also mount and use attributes for volumes, definitions
 - assigning use attributes 2-9
- resource manager parameter list 5-2
- resource managers 5-1
 - adding your own 5-4
 - example 5-4
 - creating your own 5-1
 - functions 5-1
 - at address space termination 5-1
 - at entry 5-1
 - at task termination 5-1
 - parameter list from RTM 5-2
 - key fields in 5-3
 - system resource managers provided by MVS 5-1
- resource recovery
 - See deallocation of resources in case of abend
- return attributes function of unit verification service 12-3
 - output return codes 12-15
- return codes from the USRn formatting routine 9-6
- return group ID function of unit verification service 12-2
- return indicator flag in virtual fetch parameter list
 - See VFPMRTN
- return look up value function of unit verification service 12-2
- return UCB addresses function of unit verification service 12-2
- return unit name function of unit verification service 12-2
 - output return codes 12-15
- Return unit names for a device class 12-3
- Returned storage
 - specify subpool 12-3
- RMPL
 - See resource managers, parameter list from RTM
- routing the JCL for a started task to a secondary subsystem 14-6
- running with DAT off

See DAT-off code in MVS/XA

S

- SAMPUIM 1-41
- SCHEDULR sysgen macro
 - See defining subsystems to the operating system
- SDWA
 - duplicate DAE symptom data with
 - ABDUMP 17-37
 - symptom data for DAE 17-34
 - VRA keys for DAE
 - multiple specifications of 17-38
 - symptom data specified by recovery routines 17-41
- selecting data sets for external writer processing
 - See external writer
- sending job output to the internal reader 14-3
- separating output jobs
 - See output separator routine for an external writer
- separator routine parameter list 15-9
- serialization of devices during allocation 2-1
- Service Routines, MVSCP
 - CBPADIT 1-4
 - CBPDDCT 1-4
 - CBPIDFT 1-4
 - CBPIFEAT 1-4
 - CBPIGETM 1-4
 - CBPIPARM 1-4
- Service Routines, UIMs 1-4
- SET DAE operator command
 - format 17-30
 - use 17-30
- setting a default region size via JCL 3-1
- setting default GETMAIN limits via exit routines 3-1
- setting installation defaults
 - for user region size 3-1
- setting up and using an internal reader 14-1
- SMF step initiation interface
 - See also IEFUSI exit
 - parameter list for IEFUSI 3-6, 3-7
- Specify subpool for returned storage 12-3
- SSAFF macro 11-2
 - See also subsystem affinity service
 - format 11-2
 - parameters 11-3
 - DATA 11-3
 - ENTRY 11-3
 - OBTAIN 11-3
 - SET 11-3
 - symbol 11-3
 - TCB 11-3
 - SSOB information required 11-4
- SSAFF: set/obtain subsystem affinity 11-2
- SSAT
 - See subsystem affinity table
- SSCVT
 - See subsystem communication vector table

SSIB

See SSOB information required for IEFSSREQ macro

SSOB information required for IEFSSREQ macro 11-4
in the SSIB (subsystem information block) 11-5
in the SSVS (verify subsystem extension) 11-5

SSVS

See SSOB information required for IEFSSREQ macro

START SUB= command

SUB=JES2 14-6

SUB=JES3 14-6

SUB=MSTR 14-6

START command

and the external writer 15-1, 15-12, 15-13

required for program property 4-3

to restart virtual fetch 16-6

to start virtual fetch 16-7

using a secondary subsystem 14-6

starting an external writer

See external writer cataloged procedure

starting virtual fetch 16-7

STDWTR: IBM standard output writing routine 15-2, 15-3

See also external writer

error return code 15-8

functions performed by 15-2

accessing input records 15-6

adjusting input record for output processing 15-7

closing the input data set 15-8

handling EOF on input data set 15-7

initializing the printer 15-6

reconciling control characters for output 15-6

releasing storage 15-8

logic flow 15-3

module name IEFSD087 15-2

Storage

subpool returned storage 12-3

storage flags 4-5

storage volume 2-9

See also mount and use attributes for volumes, definitions

SUB= keyword: starting a task under a specific subsystem 14-6

the subsystem affinity service 11-1

cross-memory considerations. 11-6

notes 11-6

SSAFF OBTAIN request 11-2

requirements 11-2

SSAFF SET request 11-2

requirements 11-2

using the IEFSSREQ macro 11-1

IEFSSREQ syntax 11-4

requirements for IEFSSREQ 11-4

to invoke the VERIFY SUBSYSTEM function 11-4

subsystem affinity table 11-1

See also subsystem affinity service

change of value from IPL to IPL 11-2

effect of a null SSAT 11-3

index value for a subsystem 11-1

to obtain a value from an entry 11-2

to set a value in an entry 11-2

subsystem communication vector table 10-1

subsystem definition

See defining subsystems to the operating system

subsystem options block

See SSOB information required for IEFSSREQ macro

Subsystems and the START command 14-6

subsystems defined in SYS1.PARMLIB

See defining subsystems in members of SYS1.PARMLIB

suppressing dumps automatically, by abend code 17-6

suppressing duplicate dumps (DAE) 17-26, 17-39

suppressing the display of selected messages 7-8

nonsuppressable message types 7-8

using IEAVMXIT exit routines 7-8

using MPF 7-8

using user-specified WTO exit routines 7-8

symptom data for DAE

building symptoms 17-27, 17-31, 17-35

creating and modifying 17-40

from SDWA 17-26, 17-34

in SYS1.DAE records 17-26

length requirements 17-28

matching for duplicates 17-26, 17-29

search order 17-32

symptom keys for DAE

adding to defaults 17-43

changing defaults in ADYDFLT 17-31

optional symptom keys in ADYDFLT 17-31

required symptom keys in ADYDFLT 17-31

using VRAOPT to add to defaults 17-42

using VRAREQ to add to defaults 17-42

symptom queue for DAE

creating 17-27, 17-35

definition 17-27

matching for duplicates 17-27, 17-38

updating after match processing 17-33, 17-39

symptom strings for DAE

definition for DAE 17-27

input sources for 17-41

length, increasing 17-44

matching for duplicates 17-38

minimum requirements 17-28, 17-38, 17-39, 17-42

MVS symptom strings 17-28, 17-35

RETAIN symptom strings 17-28, 17-35

stored in ADSSRNSD 17-34

stored in dump header record 17-35

symptoms for DAE

character source data 17-36, 17-37

created by DAE 17-35

created by recovery routines 17-40

default symptom keys in ADYDFLT 17-30

definition 17-27

duplicate 17-37

extraction by DAE 17-35

flag source data 17-36, 17-37

hexadecimal source data 17-36, 17-37

- minimum 17-28
- modified by recovery routines 17-40
- MVS symptoms 17-36
- optional symptom keys in ADYDFLT 17-29, 17-31
- required symptom keys in ADYDFLT 17-29, 17-31
- RETAIN symptoms 17-37
- SYSEVENT TRANSWAP**
 - See preferred storage flags in the PPT
- system dumps in MVS
 - See dump types, in MVS, SVC (system)
- system handling of duplicate subsystems 10-4
- system integrity
 - when modifying the system iii
- system log 7-8
 - controlling 7-1
 - controlling from the console 7-10
 - default operation 7-9
 - entries made via WTL macro 7-9
 - including the hardcopy log 7-8
 - information included 7-9
 - modifying the operation of 7-9
 - changing SYSOUT class: LOGCLS 7-9, 7-10
 - changing WTL count: LOGLMT 7-9, 7-10
 - example 7-10
- system module libraries
 - SYS1.AOSC5 7-6
 - SYS1.LINKLIB 7-6
 - SYS1.LPALIB 3-6, 5-4, 7-6
 - for IEFUSI 3-6
 - for PPT 4-1
 - for resource manager routines 5-4
 - for USRn system trace table entry formatters 9-5
 - for WTO/WTOR exit routines 7-6
- system operator commands
 - See also START command
 - CHNGDUMP, to change dumping options 17-3
 - DISPLAY DUMP, to determine current options 17-18, 17-19
 - DISPLAY DUMP, to display symptom data 17-13
 - DUMP, for system dumps 17-7
 - DUMPDS, to add system dump data sets 17-20
 - DUMPDS, to clear system dump data sets 17-20
 - DUMPDS, to define system dump data sets 17-1
 - DUMPDS, to delete system dump data sets 17-5, 17-20
 - MOUNT, assigning volume attributes 2-8, 2-9, 2-11
 - MOUNT, required for program property 4-3
 - WRITELOG, controlling the system log 7-10
- system parameter library
 - See SYS1.PARMLIB
- system trace facility
 - See PTRACE macro
- system trace table entry, formatting

- See formatting a USRn trace table entry
- system trace table entry, user-defined
 - See USRn system trace table entry
- SYS1.AOSC5, library for WTO/WTOR exit module 7-6
- SYS1.DAE data set
 - creating 17-32
 - DCB attributes 17-33
 - description 17-32
 - matching 17-26
 - protection 17-33
 - sample JCL for creating 17-33
 - updating by DAE 17-26, 17-33
 - using existing data set 17-33
- SYS1.LINKLIB
 - See system module libraries
- SYS1.LPALIB
 - See system module libraries
- SYS1.MACLIB 10-3
- SYS1.NUCLEUS, the DAT-off nucleus 6-1
- SYS1.PARMLIB
 - ADYSETxx members 17-29
 - COMMNDxx member, to start virtual fetch 16-7
 - IEASYSxx member, to select parmlib members at IPL 10-2
 - IEASYSxx members, for system parameters 2-7
 - IEASYSxx members, to control the system log 7-9
 - IECIOSxx member, for hot I/O control 13-3
 - IEFSSNxx members, to define subsystems 10-1, 10-2
 - LNKLSTxx member for external writer routines 15-4, 15-10, 15-13
 - LNKLSTxx members, for resource managers 5-4
 - MPFLSTxx members, to suppress messages 7-8
 - SCHEDxx member 2-7
 - VATLSTxx members, for volume attributes 2-2, 2-8, 2-9

T

- tailoring dumps by type 17-7
- tailoring dumps in MVS and MVS/XA
 - See MVS dumping services
- TCAM Message Control Program 4-9
- TCB subsystem affinity
 - See subsystem affinity service
- the hot I/O detection table 13-1
- the virtual fetch service
 - See virtual fetch
- tips on using the preferred storage flags 4-6
- TTE for user data in system trace table
 - See USRn system trace table entry
- TTE, formatting
 - See formatting a USRn trace table entry

U

- UCA 1-3
- UIM
 - communication area 1-2
 - considerations in coding 1-12
 - CPVT 1-3
 - entry logic 1-13
 - exit logic 1-13
 - Macros 1-14
 - CBPYDIP 1-14
 - CBPZCPVT 1-14
 - CBPZDCP 1-14
 - CBPZDFP 1-14
 - CBPZDIAG 1-14
 - CBPZFCP 1-14
 - CBPZGETM 1-14
 - CBPZIODV 1-14
 - CBPZITRH 1-14
 - CBPZLOG 1-14
 - CBPZLOGR 1-14
 - CBPZPCP 1-14
 - CBPZPPDS 1-14
 - CBPZUCA 1-14
 - MVSCP vector table 1-3
 - Processing 1-2
 - End-of-data 1-2
 - restrictions 1-12
 - UCA 1-3
- UIM communications area 1-3
- UIM Data Areas
 - control blocks 1-3
 - external to UIMs 1-3
 - CPVT 1-3
 - Configuration program vector table 1-3
 - IODV 1-3
 - IODEVICE internal text record 1-3
 - UCA 1-3
 - UIM communications area 1-3
- UIM Macros 1-38
- UIM message identifiers 1-15
- UIM Service Routines
 - CBPADIT 1-4
 - CBPDDCT 1-4
 - CBPIDFT 1-4
 - CBPIFEAT 1-4
 - CBPIGETM 1-4
 - CBPIPARM 1-4
- UIMs
 - a Sample UIM 1-1, 1-2
 - general logic of processing a UIM 1-1, 1-2
 - how MVSCP uses 1-1, 1-2
 - how MVSCP uses UIMs 1-1, 1-2
 - IBM-supplied 1-39
 - List of (partial) 1-39
 - macros required 1-38
 - CBPZDIAG 1-38
 - CBPZLOG 1-38
 - CBPZPPDS 1-38
 - IOSDDT 1-38
 - IOSDMLT 1-38
 - naming 1-38
 - customer-supplied 1-38
 - IBM-supplied 1-38
 - purpose 1-1, 1-2, 1-38
 - residence 1-38
 - concatenated to LINKLIB 1-38
 - LINKLIB 1-38
 - sample 1-41
 - UIM Data Areas 1-1, 1-2
 - UIM Service Routines 1-1, 1-2
 - using the sample UIM 1-41
 - in SYS1.SAMPLIB(SAMPUIM) 1-41
 - writing 1-1, 1-2, 1-38
 - writing a UIM 1-1, 1-2
- UIMs in LINKLIB 1-38
- UIMs Service Routines 1-4
- Unit Information Modules
 - coding a DUMMY device 1-42
 - considerations in coding 1-12
 - diagnosis 1-15
 - CBPZDIAG macro 1-15
 - CBPZLOG macro 1-15
 - CBPZLOGR macro 1-15
 - CBPZPPDS macro 1-15
 - IBM-supplied 1-39
 - List of (partial) 1-39
 - message ids 1-15
 - naming 1-38
 - customer-supplied 1-38
 - IBM-supplied 1-38
 - processing logic 1-11
 - residence 1-38
 - concatenated to LINKLIB 1-38
 - LINKLIB 1-38
 - sample 1-41
 - sample JCL 1-42
 - sample MLT 1-43
 - using the sample UIM 1-41, 1-42
- unit name is a look-up value function of unit verification service 12-3
- Unit names
 - for a device class 12-3
- Unit Verification
 - specify subpool for returned storage 12-3
 - unit names for a device class 12-3
- unit verification functions
 - as input to unit verification functions 12-2, 12-3
 - attributes returned by IEFGB4UV, IEFAB4UV and IEFEB4UV 12-15
 - check groups 12-1
 - check units 12-2
 - convert device type to look-up value 12-2
 - no validity bit 12-3
 - return attributes of unit name 12-3
 - return group ID 12-2
 - return look-up value 12-2
 - return UCB addresses 12-2
 - return unit name 12-2

- unit name is a look-up value 12-3
- Unit verification routine
 - IEFAB4UV 12-4
 - IEFEB4UV 12-4
 - IEFGB4UV 12-4
- Unit verification service 12-1
 - explanation 12-1
 - for authorized programs: IEFGB4UV or IEFAB4UV 12-1
 - for problem programs: IEFEB4UV 12-1
 - input to and output from 12-5
 - FLAGS parameter field 12-5
 - parameter list required 12-5
 - problem program callers 12-1
 - purpose 12-1
- UNITNM 1-42
- updating entries in the PPT
 - See program properties table
- updating modules managed by virtual fetch
 - See virtual fetch, refresh processing
- updating SYS1.DAE 17-26, 17-39
- updating the master job control language data set 8-1
- updating the PPT 4-9
- use attribute, for volumes
 - See mount and use attributes for volumes
- user dumps in MVS
 - See dump types, in MVS, ABEND (user)
- user-assigned group names
 - See esoteric group names
- user-specified WTO exits
 - capabilities 7-5
 - routine's function 7-4
 - uses for 7-4
- using a message-routing/processing exit routine 7-3
- using the block character routine 15-11
- using the PTRACE macro 9-2
- using virtual fetch service
 - See virtual fetch
- USRn system trace table entry 9-1
 - contents 9-1
 - continuation information in 9-3
 - example 9-4
 - sample of code for 9-7

V

- V = R area and PPT 4-4
- VATLST
 - See volume attribute list
- verification of devices 12-1
- VERIFY SUBSYSTEM function 11-1
 - See also subsystem affinity service, using the IEFSSREQ macro
- VFCB
 - See virtual fetch
- VFDE
 - See virtual fetch
- VFIN00 - VFIN99

- See The Virtual Fetch Service, DD statements needed
- VFBM
 - See virtual fetch service, parameter list
- VFBMRTN 16-15
- VFVT, created by virtual fetch build function 16-6
- VFWK, created by virtual fetch build function 16-6
- VIO data sets
 - See virtual fetch
- virtual fetch 16-1
 - considerations when using 16-9
 - DD statements needed in cataloged proc 16-7
 - example 16-7
 - names range from VFIN00 to VFIN99 16-7
 - defined 16-1
 - dumping the managed modules 16-11
 - functions 16-1
 - BUILD, FIND, and GET requests 16-3
 - build, using CSVVFNDE 16-6
 - find function, using CSVVFNDE and CSVVFSCH 16-6
 - get, using CSVVFGTE and CSVVFTCH 16-6
 - initialization by CSVVFCRE 16-2
 - hash table of directory entries 16-2
 - initialization 16-2-16-3
 - parameter list for build, find, and get functions 16-9
 - format and contents 16-10
 - notes on use 16-10, 16-15
 - programming considerations 16-10
 - conventions 16-11
 - for a BUILD request 16-13
 - for a FIND request 16-13
 - for a GET request 16-14
 - parameters 16-10
 - refresh processing 16-6, 16-8
 - using CSVVFCRE and CSVVFRSH 16-8
 - return codes 16-2
 - from a FIND request 16-14
 - from BUILD request 16-13
 - from CSVVFCRE for initialization 16-2
 - if virtual fetch address space terminates 16-6
 - in parameter list field VFBMRTN, from a GET request 16-15
 - starting 16-1
 - using a cataloged procedure 16-1
 - VIO data set 16-1
 - renewed by refresh processing 16-6, 16-8
- virtual fetch build function 16-6
- virtual fetch find function 16-6
- virtual fetch get function 16-6
- virtual fetch initialization function 16-2
- virtual storage management
 - figure showing default limits set by VSM 3-5
 - how VSM uses region size and GETMAIN limits 3-9
 - limiting user region size 3-1
 - based on IEALIMIT values 3-2, 3-3
 - based on IEFUSI values 3-2
 - example using IEFUSI values 3-8

- in the absence of an IEFUSI routine 3-2
- parameter list 3-6
- result of using default values 3-3
- setting GETMAIN limits 3-4
- volume attribute list 2-2, 2-8
- VRA keys in SDWA for DAE
 - description 17-41
 - multiple specifications 17-38
 - user-defined keys 17-42
- VRADAE key and DAE processing 17-40, 17-42
- VRADATA macro 17-41
- VRAMINSC key and DAE processing 17-42
- VRAMINSL key and DAE processing 17-42
- VRAOPT key and DAE processing 17-42
- VRAREQ key and DAE processing 17-42
- VSAM interface, for internal reader 14-2
- VSM parameter list
 - flag word bit meanings 3-7
 - parameter list for IEFUSI
 - flag byte bit meanings 3-7

W

- write-to-operator macros 7-2
 - as messages to the console 7-2
- WRITELOG operator command 7-10
- writing an ITRF0n7F TTE formatting routine 9-5
- writing an output separator routine 15-10
- writing your own output writing routine 15-3
- WTO/WTOR exits 7-1
 - and MCS 7-3
 - at IEAVMXIT 7-3, 7-4
 - at IEECVXIT 7-3
 - at user-specified exits 7-3, 7-4
 - bypassed by the system 7-3

- descriptor codes 7-3
 - function 7-3
- inserting into the control program 7-5
 - after IPL time 7-6
 - before IPL 7-6
 - before sysgen 7-6
 - IEAVMXIT routine 7-6
 - user-specified WTO exit routine 7-6
 - routine's function 7-3, 7-4
 - routing codes 7-2, 7-3
- WTO/WTOR macros
 - See write-to-operator macros

X

- XWTR cataloged proc
 - See external writer cataloged procedure
- xx value on SET DAE operator command 17-30

Z

- ZAP
 - modifying entries in IEAVTRML 5-4
 - to modify MSTRJCL statements 8-2

Numerics

- 31-bit addressing
 - See extended addressing considerations



GC28-1152-2

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Cut or Fold Along Line

Fold and tape

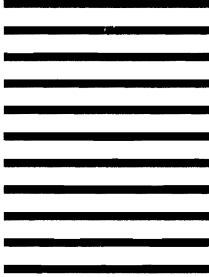
Please Do Not Staple

Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO Box 390
Poughkeepsie, New York 12602



Fold and tape

Please Do Not Staple

Fold and tape



Printed in U.S.A.





Printed in U.S.A.

