IBM z/VSE
6.1

System Macros User's Guide

SC34-2709-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 223.

This edition applies to Version 4 Release 3 of IBM z/Virtual Storage Extended (z/VSE), Program Number 5609-ZV4, and to all subsequent releases and modifications until otherwise indicated in new editions

This edition replaces SC34-2709-00.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

```
IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

You may also send your comments by FAX or via the Internet:

```
Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456
```

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures ix

Tables xiii

About This Publication xv
Who Should Use This Publication xv
Where to Find More Information xv

Summary of Changes xvii

Chapter 1. Introduction 1
Using a Macro in a Program 1
Types of Macros 1
Register Usage 2
Macro Format 2

Chapter 2. Data Management Concepts 5
I/O-Related Hardware Characteristics 6
I/O Devices 6
Disk Volume 7
Tape Volume 10
File, Extent, and Volume Relationship 11
Record, Block, and Control Interval 12
The Records of a File 12
Control Interval 15
Blocking and Deblocking of Logical Records 16
Device-Dependent Record Formats 16
Organization of Records in a File 20
Choosing the Right Access Method 21
Sequential Versus Direct Processing 22
Level of Support 22
Sequential Access Method (SAM) 23
VSE/VSAM 23
Direct Access Method (DAM) 23
Indexed Sequential Access Method (ISAM) 24
Support of Access Methods by Programming Languages 24
Volume and File Labels 25
Volume Labels 26
File Labels 26
Volume Organization 26
Disk Volume 27
Tape Volume 32
The Input/Output Control System (IOCS) 34
DTFxx Macro 34
Logic Module Generation (xxMOD) Macro 37
Logical IOCS Versus Physical IOCS 39

Chapter 3. Defining and Processing a File with SAM 41
Defining the Characteristics of a File 41

File Type (TYPEFLE)..... 42
Record Format (RECFORM)..... 42
Record Size (RECSIZE)..... 42
I/O Area Definition (IOAREA)..... 43
I/O Area Length (Block Size BLKSIZE)..... 43
I/O Register Specification (IOREG)..... 43
Work Area Specification (WORKA)..... 44
Error Handling (ERROPT, WLRERR, and ERREXT)..... 44
End-of-File Exit (EOFADDR)..... 45
Opening a File for Processing..... 45
Reading (GET) and Writing (PUT) of Data..... 47
Obtaining a Record for Processing (GET)..... 47
Storing a Record after Processing (PUT)..... 48
Processing Blocked Records Selectively..... 49
Processing a Work File..... 50
Opening the File..... 50
Processing the File Sequentially..... 51
Processing the Records of the File Selectively..... 52
Retaining or Deleting a Work File..... 54
Requesting a Non-Data Device Operation..... 55
Closing the File for Processing..... 55
IOCS Request Macros Used with Declarative Macros..... 55

Chapter 4. Processing a Disk File with SAM..... 57
Opening the File..... 57
Processing of Labels..... 58
Processing for OPEN..... 58
Processing on End of Volume..... 59
Processing on End of File..... 59
User-Standard Labels..... 59
Returning Control to SAM..... 60
Processing an Update File..... 60
Coding an Error-Processing Routine..... 61
Wrong-Length Error..... 62
Other Errors..... 63
Closing a File and Processing for End of Volume..... 63
Process a File Residing on an FBA Disk..... 64

Chapter 5. Processing a Tape File with SAM..... 65
Processing of Labels..... 65
Output File..... 65
Input File..... 67
Unlabeled File..... 68
American National Standard Labels..... 70
Return Control to SAM..... 70
End of Volume for a Multi-Volume File..... 71
Tape File Extension..... 71
Processing for IBM Standard Labels..... 72
Processing for User-Standard Labels..... 72
Coding an Error-Processing Routine..... 72
Wrong-Length Error Processing Considerations..... 74
Other Error-Processing Considerations..... 74
Non-Data Operations..... 74
Rewind and Tape-Movement Functions..... 75
Spacing Over a Logical Record..... 75
Synchronizing the Hardware Buffer..... 76
User Interface for Tape OPEN, CLOSE, and End-of-Volume..... 76

Access-Protection for an ASCII Tape....77

Chapter 6. Processing a Unit Record File with SAM....79
Processing a Punched Card File....79
Programming for Associated Files....79
Updating a Record....81
Optical-Mark-Read and Read-Column-Eliminate Modes....81
End-of-File Handling....84
Error Handling....84
Hints for Programming....84
Non-Data Device Operations....85
Processing a Printer File....88
Associated File on an IBM 3525....88
Printer Overflow....88
Printer Controls....88
Programming for Output to an IBM 4248....91
Error Handling....92
Processing a Console File....93

Chapter 7. Processing a Device-Independent System File with SAM....95
Restrictions for DTFDI Processing....95
Record Size....95
Error Handling....96
Wrong-Length Record Errors....96
Irrecoverable I/O Error....96
End-of-File Handling....97

Chapter 8. Requesting Control Functions....99
Program Loading....99
The Load Request....99
Load Request for a Phase in the SVA....100
Fast Loading of Frequently Used Phases....100
Virtual Storage Control....100
Fixing and Freeing a Page in Processor Storage....101
Determining the Run Mode of a Program....102
Extracting Partition-Related Information....102
Reducing the Number of Page Faults....102
Allocating Virtual Storage Dynamically....103
Program Communication....103
Assigning and Releasing an I/O Unit....105
Explanation of Function Codes in Detail....106
Timer Services....108
Time-of-Day Clock....108
Interval Timer....109
Linkage to User Exit Routines....110
Interval-Timer User Exit....110
Abnormal-End User Exit....112
Program-Check User Exit....112
Operator-Communication User Exit....113
Ending a Job Step....113
Program Linkage....113
Linkage Macros....115
Loading a Forms-Control Buffer....117
Multitasking Functions....118
Subtasking and I/O Requests....118
Starting (Attaching) a Subtask....118
Ending (Detaching) a Subtask....120

Task-to-Task Communication within Partition ... 120
Resource Protection ... 122
Resource-Share Control ... 125
DASD Record Protection (Track Hold) ... 126
Shared Modules and Files ... 128
Multitasking Sample Program ... 129
Requesting Storage Dumps ... 132
The DUMP Macro ... 132
The JDUMP Macro ... 132
The PDUMP Macro ... 132
The SDUMP and SDUMPX Macros ... 132
Requesting Volume and Device Characteristics ... 133
Retrieving Volume and Device Characteristics ... 133
Obtaining the Track Balance of a Device ... 133
Obtaining the Track Capacity of a Device ... 134
Requesting System Information ... 134
Writing and Deleting Messages (WTO, WTOR, and DOM Macros) ... 135
Routing the Message ... 135
Altering Message Text ... 135
Writing a Multiple-Line Message ... 136
Deleting Messages Already Written ... 136
WTO, WTOR, DOM Usage Examples ... 136
Example of an LBSERV MOUNT Request ... 138
Library Access for Application Programs ... 139
Storage Requirements ... 139
Record I/O ... 140
Librarian Control Block ... 140
Accessing or Updating Librarian Member Data ... 140
Library Access Functions ... 141
Return Code Conventions ... 142
Record Formats ... 143
Processing Sequence of Sublibrary Chains ... 143
Register Usage ... 144
Librarian Exits ... 144
Library Access Request Sequence ... 147
Cross-Partition Communication ... 148
Identification of Communication User ... 149
Defining a Communication Path ... 150
Defining a Specific Connection ... 150
Defining an Open-Ended Connection ... 152
Data Transmission ... 152
Sending and Receiving Data ... 152
Receiving Data ... 158
The REPLY Function ... 160
Clearing a Pending SEND/SENDR Request on the Sender's Side ... 161
Clearing a Pending SEND/SENDR Request on the Receiver's Side ... 162
Disconnecting from a Communication Path ... 163
Terminating XPCC Usage ... 164
Abnormal End Processing ... 165
Compressing and Expanding Data ... 165
Compression and Expansion Dictionaries ... 166
Compression Processing ... 167
Expansion Processing ... 167
Dictionary Entries ... 168
Compression Dictionary Entries ... 168
Expansion Dictionary Entries ... 170
Dictionary Restrictions ... 171
Other Considerations ... 171

Compression Dictionary Examples....172
Expansion Dictionary Example....175
Building the CSRYCMPS Area....176
Determining if the CSRCMPSC Macro Can Be Issued on a System....178
Compression/Expansion Examples....178
Example 1....178
Example 2....179
Example 3....179

Appendix A. Assemble and Link-Edit Programs Using IOCS....181
Cataloging Assembled DTFxx and xxMOD Macros....181
Assembling and Cataloging IOCS Modules....181
IOCS Sample Program....182
Assemble the DTFs and Logic Modules Separately....184
Comparison of the Three Possible Methods....186

Appendix B. Direct Access Method (DAM)....189
Defining the File....189
Processing the File....191
Opening the File....192
Creating a File and Adding of Records to a File....192
Locating a Record....193
Locating a Free Space....197
Reading and Writing a Record....197
Non-Data Device Operation....207
Error Handling....207
Closing the File....210

Appendix C. Processing a File with Physical IOCS (PIOCS)....211
Opening the File....211
Disk Volumes – Output....212
Disk Volumes – Input....213
Processing of User Labels and Extent Information....213
Reading and Writing of Records....214
The Command Control (I/O-Request) Block....215
The Execute Channel Program (EXCP) Macro....215
The WAIT Macro....215
Additional Macros....216
Forcing an End-of-Volume Condition....216
Closing the File....217
Hints for Programming....217

Appendix D. Using System Control Macros in Reenterable Programs....221

Notices....223
Programming Interface Information....224
Trademarks....224
Terms and Conditions for Product Documentation....224

Accessibility....227
Using Assistive Technologies....227
Documentation Format....227
Glossary....229

Index....265

Figures

1. Schematic Example of the Processing of a Macro....1
2. Relation of Data Management to Devices and Application Program....5
3. Cylinders and Tracks on a Volume of a CKD or ECKD Disk Device....8
4. Track and Record Formats for CKD and ECKD Devices....9
5. Blocks as Stored on a Volume of an FBA Device....10
6. File, Extent, and Volume Relationships....12
7. Logical Record....13
8. Stored Record of Variable Length....13
9. Format of a Spanned Record....14
10. Block and Blocking Factor....15
11. Layout of a Control Interval....16
12. Hardware-Dependent Format of a CKD or ECKD Recordl....17
13. Records of Fixed and Variable Length (Without Key Area) on a CKD Disk....18
14. Spanned Records on CKD or ECKD Devices....19
15. The Records of a File with Keys....21
16. Locating a Record Via an Index....21
17. Disk Volume Organization – Concepts....27
18. General VTOC Format for a Disk Volume....28
19. Format-1 Label Overview....28
20. Disk Volume Layout: One File plus VTOC....29
21. Disk Volume Layout: Files with User-Standard Labels - Part 1....30
22. Disk Volume Layout: Files with User-Standard Labels - Part 2....31
23. Disk Volume Layout with VSAM Data Spaces....32

24. Tape Volumes with IBM and User-Standard Labels....33
25. Tape Volumes with Nonstandard Labels....33
26. Tape Volume with Unlabeled Files....34
27. Relationship Between IOCS Macros and Logic Modules....34
28. Sample DTFMT Macro....35
29. Relationship Between Request Macro, DTF Table, and I/O Device Assignment....37
30. Example of a Subset/Superset Naming Chart....39
31. Coding Example for the Use of a Work Area....44
32. Coding Example for Opening and Accessing a File....46
33. Example for Defining a Work File on Disk....50
34. Example of POINTS Macro with Work File Processing....53
35. Coding Example for Processing an Update File....61
36. Example of a Combined File....81
37. OMR Data and Format-Descriptor Example....83
38. Print Channel to Card Row Correspondence on an IBM 3525....87
39. Example for Using the LOAD Macro with a Local Directory List....100
40. PFIX and PFREE Example....102
41. Parameter List Generated by the ASPL Macro....106
42. Example for a Device Assignment....108
43. Example of Using the Interval Timer Exit....111
44. Example of Multi-Task Linkage to a Common Exit Routine....111
45. Example of an Exit Routine Processing a Program Check....112
46. Example for Using the Macros CALL, SAVE, and RETURN....116
47. Example for Loading an Alternate FCB....117
48. Waiting for Preferred and Secondary Events....121

49. Use of the POST Macro..... 122

50. Sharing a Resource in a Common Subroutine..... 123

51. Sharing a Routine Which Is Part of One Task..... 124

52. Sharing a Resource in Different Subroutines..... 124

53. Example of the UNLOCK Macro..... 125

54. Using the Track Hold Facility..... 128

55. Multitasking Sample Program..... 129

56. Example of Single-Line WTO..... 136

57. Example of Multiple-Line WTO..... 137

58. Example of a Backward-Compatible WTO..... 137

59. Example of WTO for Command Responses..... 137

60. Example of WTOR..... 137

61. Example of Application Control of Message Deletion (DOM)..... 137

62. Library Access Request Sequence..... 147

63. Job Stream for Assembling and Cataloging IOCS Modules..... 182

64. IOCS Sample Program - Source Code for Common Assembly..... 183

65. Program, DTF Tables, and Logic Modules in Storage..... 184

66. Source Code Modifications for a Separate Assembly..... 185

67. Contents of Record 0 with Capacity-Record..... 193

68. Track Reference Fields – Physical Track Addressing..... 195

69. Track Reference Fields – Relative Track Addressing..... 196

70. Prime Data Record and Related Overflow Records..... 200

71. Relationship Between Source Program and Job Control I/O Assignment..... 211

72. Relationship Between the PIOCS Macros (No DTFPH Used)..... 216

73. Relationship Between DTFPH and Other PIOCS Macros..... 217

74. Example of Channel Programming a File-Protected CDK DASD File.....219

75. Dynamically Acquiring Storage and Addressing this Storage.....222

Tables

1. Available Access Methods and their Scope of Support....22

2. Access Methods.... 24

3. Volume and File Labels as Supported by a VSE System.... 25

4. SAM DTFxx and xxMOD Macros by Supported Device Classes....41

5. IOCS Request Macros Used with SAM Declarative Macros....55

6. Sequence of GET/CNTRL/PUT Macros for Associated Files....80

7. XPCCB control block input 149

8. XPCCB control block output.... 149

9. CONNECT request XPCCB input fields.... 150

10. CONNECT request XPCCB output fields....151

11. SEND request - Input fields in the XPCCB used on the sender's side. 154

12. SEND request - Output fields in the XPCCB used on the sender's side. 154

13. SEND request - Output fields in the XPCCB used on the receiver's side.... 155

14. IJBXADR 8-byte fields.... 155

15. REPLY request - Input fields in the XPCCB used on the sender's side. 156

16. REPLY request - Output fields in the XPCCB used on the sender's side. 156

17. REPLY request - Output fields in the XPCCB used on the receiver's side.... 156

18. SENDI request - Input fields in the XPCCB used on the sender's side. 157

19. SENDI request - Output fields in the XPCCB used on the sender's side. 157

20. SENDI request - Output fields in the XPCCB used on the receiver's side....158

21. RECEIVE request - Input fields in the XPCCB used on the receiver's side. 158

22. RECEIVE request - Output fields in the XPCCB used on the receiver's side. 159

23. RECEIVE request - Output fields in the XPCCB used on the sender's side....159

24. REPLY request - Input fields in the XPCCB used on the replier's side. 160

25. REPLY request - Output fields in the XPCCB used on the replier's side. 160

26. REPLY request - Output fields in the XPCCB used on the sender's side..... 160

27. CLEAR request - Input fields in the XPCCB used on the sender's side. 161

28. CLEAR request - Output fields in the XPCCB used on the sender's side. 161

29. CLEAR request - Output fields in the XPCCB used on the receiver's side..... 161

30. PURGE request - Input fields in the XPCCB used on the receiver's side. 162

31. PURGE request - Output fields in the XPCCB used on the receiver's side. 162

32. PURGE request - Output fields in the XPCCB used on the sender's side..... 162

33. DISCONNECT request - Input fields in the XPCCB 163

34. DISCONNECT request - Output fields in the XPCCB 163

35. DISCONNECT request - Output fields in the XPCCB used on the partner's side..... 163

36. TERMINATE request - Input fields in the XPCCB used by the system..... 164

37. TERMINATE request - Output fields in the XPCCB used by the system..... 164

38. TERMQSCE request - Output fields in the XPCCB, which have CONNECTs for terminating applications pending (TERMQSCE only)..... 165

About This Publication

This publication is intended as a guide for programmers using the macro support available with IBM z/VSE. These macros can be used in application programs (or routines of such programs) written in assembler language.

The publication describes the two types of macros that z/VSE offers: data management or input/output (IOCS) macros and control program macros.

Who Should Use This Publication

This publication is mainly intended for programmers writing application programs in assembler language.

Where to Find More Information

The macros mentioned in this book are described in detail in:

- z/VSE System Macros Reference

To efficiently use this publication, you should be familiar with other IBM publications that provide information that you might need:

- *High Level Assembler for z/OS & z/VM & z/VSE Programmer's Guide*
- *High Level Assembler for z/OS & z/VM & z/VSE Language Reference*
- *Principles of Operation* publication for your processor.
- z/VSE Guide to System Functions

z/VSE IBM Documentation

IBM Documentation is the new home for IBM's technical information. The z/VSE IBM Documentation can be found here:

https://www.ibm.com/docs/en/zvse/6.2

You can also find VSE user examples (in zipped format) at

https://public.dhe.ibm.com/eserver/zseries/zos/vse/pdf3/zVSE_Samples.pdf

Summary of Changes

This publication has been updated to reflect terminology, maintenance, and editorial changes.

Chapter 1. Introduction

To write the source code of a program, you would normally use a high-level programming language supported by z/VSE. However, performance considerations may require you to code one or more of your program's paths in assembler language. The macros described in z/VSE System Macros Reference provide the necessary means to communicate with z/VSE on this level.

You can use these macros repeatedly in any of your programs. Their use reduces the possibility of programming errors.

Note: The publication also uses the short form VSE to mean z/VSE or its predecessor, VSE/ESA.

Using a Macro in a Program

A macro, a single statement, causes the assembler to generate a sequence of instructions. This sequence is determined by a macro definition stored in a sublibrary under the name of the macro's operation code. The sublibrary must be accessible by the assembler.

Note: Starting with VSE/ESA 2.1. the DOS/VSE Assembler is no longer part of VSE. It has been replaced by the High Level Assembler (full name: IBM High Level Assembler for MVS & VM & VSE) which must be called with a // EXEC ASMA90.... JCL statement.

For related planning and migration information, refer to z/VSE Planning.

The High Level Assembler itself is described in the following publications:

- *High Level Assembler for z/OS & z/VM & z/VSE Programmer's Guide*
- *High Level Assembler for z/OS & z/VM & z/VSE Language Reference*

Figure 1 on page 1 shows how a macro is used in a source program, how the assembler generates the desired sequence of code (also called a macro expansion), and how it inserts this sequence into your source program.

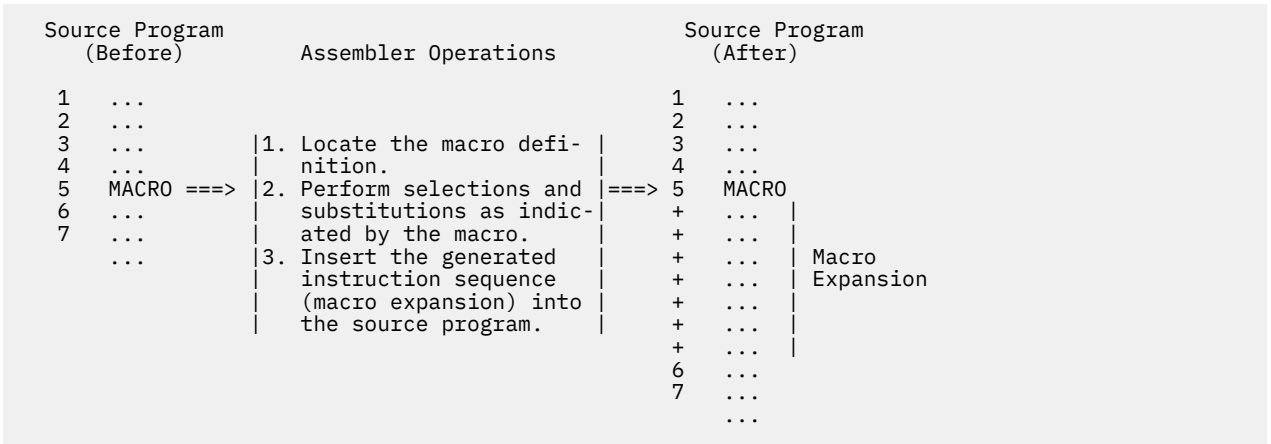


Figure 1. Schematic Example of the Processing of a Macro

Types of Macros

Functionally, the available macros fall into two categories: control program macros and data management or IOCS (input/output control system) macros.

- Control program macros

  They enable you to make use of certain system services provided by VSE. Examples of these services are the timer services or the multitasking functions. These macros are discussed in Chapter 8, "Requesting Control Functions," on page 99.

- IOCS macros

  They are a bit more complex. To use them, you should be familiar with the concepts of data management as described in Chapter 2, “Data Management Concepts,” on page 5. The topic gives you an overview of hardware-related input/output considerations; it tells how the IOCS macros relate to these hardware-related considerations.

For a full description of these two categories of macros available from IBM, refer to z/VSE System Macros Reference.

Before you go deeper into the subject of using macros in your program, familiarize yourself with a few coding conventions.

Register Usage

Registers 2 through 12 are available for general use. However, the PUTR (PUT with Reply) macro makes use of register 2. General registers 0, 1, 13, 14, and 15 are available to your program only under certain conditions.

The following paragraphs describe the general uses of these registers by IOCS, but the description is not meant to be all inclusive. For more information on subroutine linkage through registers, refer to “Linkage Registers” on page 113. Certain applications might require different registers.

- Registers 0, 1, and 15

  IBM-supplied macros use these registers to pass parameters and return codes. Therefore, the registers may be used without restriction only for immediate computations.

- Register 13

  System routines, and also IOCS routines, use this register as a pointer to a 72-byte save area. When using the CALL, SAVE, or RETURN macro, you can set the address of the save area at the beginning of each phase of your program, and leave it unchanged thereafter. However, if reentrant, read-only code is shared among tasks, register 13 must contain the address of another save area to be used by that code each time the code is used by another task.

- Registers 14 and 15

  IOCS uses these registers for linkage without saving their contents. If you use the registers, either save their contents (and reload them later) or finish with these registers before IOCS uses them.

  Not all logic modules use standard save area conventions. Therefore, if you use a read-only logic module (supplying a module save area) in a subroutine, the save area back-chain pointer can get lost.

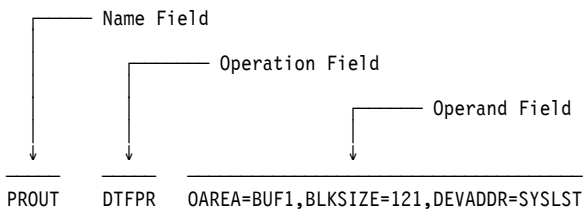
- Floating Point Registers

  If your program uses floating-point registers in a subroutine, ensure that this subroutine:

  1. Saves their contents when it receives control.
  2. Restores their contents when it returns control.

Macro Format

Following is an example of a macro showing the various fields (refer also to the documentation of the High Level Assembler).



Comments can be included just as in assembler instructions. However, a macro without an operand requires a comma preceding the comment. Example:

```
CANCEL   ,  NO USE TO CONTINUE PROCESSING
```

The name field in a macro may contain a symbolic name. Some macros, CCB or TECB for example, require a name.

The operation field must contain the mnemonic operation code of the macro.

The operands in the operand field are of positional format, of keyword format, or they are mixed.

Positional Format

If a macro has this format, its operands must be coded in a given order. Each operand, except the last, must be followed by a comma; no embedded blanks are allowed. If an operand is to be omitted and subsequent operands are to be included, a comma must be inserted to indicate the omission. No such commas need be included after the operand coded last. For example, the macro GET (for a read-data operation) uses the positional format. To read from a file named CDFILE, requesting the retrieved data to be moved to a work area named WORK, you would code:

```
GET CDFILE,WORK
```

Keyword Format

An operand written in keyword format has the form as shown by the example below:

```
LABADDR=MYLABELS
```

where LABADDR is the keyword and MYLABELS is the specification (or value).

The keyword operands of a macro may appear in any order, and any operands not required may be omitted. Different keyword operands may be written on the same line, each followed by a comma except for the last operand of the macro. They may be written on separate lines as shown in Figure 28 on page 35.

Mixed Format

The operand list contains both positional and keyword operands. The keyword operands can be written in any order, but they must be written to the right of any positional operand in the macro.

Continuation on the Next Line

Column 72 must contain a continuation character (any non-blank character) if the operands fill the operand field and overflow onto another line.

Chapter 2. Data Management Concepts

If you are already familiar with the data management functions of an operating system, you might want to skip some of the sections. To familiarize yourself with the hardware-related aspects of programming for I/O under VSE, read:

- “File, Extent, and Volume Relationship” on page 11.
- “Record, Block, and Control Interval” on page 12.

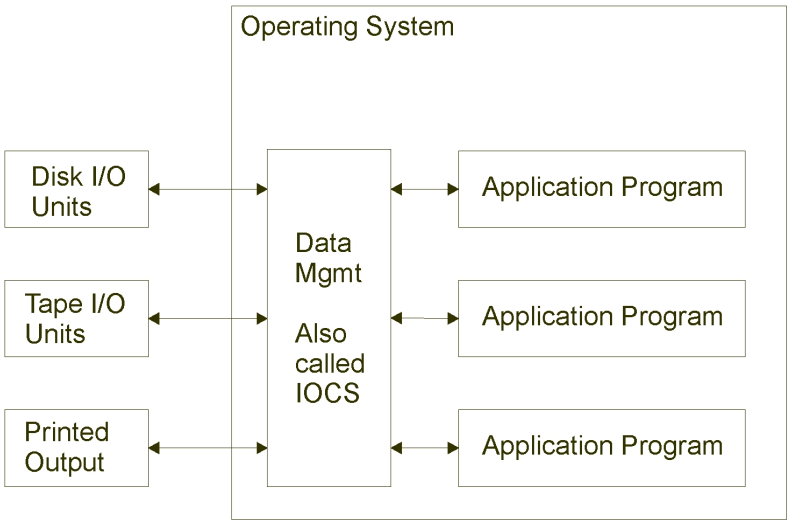
The term data management collectively describes those operating system functions that control the flow of data from an input/output (I/O) device to the processing program's data buffer and vice versa. It describes the functions that enforce data storage conventions.

You as a programmer are primarily concerned with the input and output requirements of your application. Some of these requirements are:

- Does the program update the input file and will the updated file become the input file for future processing?
- Is the data subject to inquiries?
- Is it faster to process the records of the file one after the other or by direct access?

The answers to these and many other questions determine, which of the available methods of file access to choose. They determine how to organize your data or which storage device (disk or tape, for example) is best suited for the application.

Before going into the actual data-management discussion, have a look at Figure 2 on page 5.



Legend: ===> = Flow of data

Figure 2. Relation of Data Management to Devices and Application Program

Figure 2 on page 5 shows that data management, a set of operating system routines called IOCS, is primarily concerned with:

- Reading data

  This involves the transfer of data from a device to an area in virtual storage. It involves locating logical records, one after the other, if the unit of data transfer comprises two or more logical records.
- Writing data

  This involves the transfer of data from virtual storage to an output device. It may involve the grouping logical records into blocks if the unit of data transfer comprises two or more logical records.

This section discusses the data management concepts of VSE by major topics as follows:

- I/O-related hardware characteristics of interest to an application programmer – Section "I/O-Related Hardware Characteristics" on page 6.
- The general organization of data stored on disk – Section "File, Extent, and Volume Relationship" on page 11.
- A summary of criteria for selecting the best suited access method – Section "Choosing the Right Access Method" on page 21.
- How the system identifies data stored on volumes of auxiliary storage – Section "Volume and File Labels" on page 25.
- An overview of the IBM provided I/O macros and their use in an application program – Section "The Input/Output Control System (IOCS)" on page 34.

I/O-Related Hardware Characteristics

This section gives an overview of those hardware characteristics that are of interest to an application programmer. It discusses the characteristics of volumes for auxiliary storage – disk volumes and tape volumes.

I/O Devices

Following is a list of I/O device classes supported by VSE:

- Serial only devices:
- Card readers and card punches
- Printers
- Magnetic tape drives
- Optical and magnetic character readers
- Consoles
- Direct access (disk) devices

The characteristics of the available devices may well influence your choice of I/O media for your application. These characteristics therefore deserve a closer look.

Characteristics of Serial Devices

They all have one application-related characteristic in common. Your program can either read from the device (for data input) or write to the device (for data output). It can either read from or write to the device only one record after the other. Nearly all of the above mentioned device classes can be used for both input and output, but not at the same time. Exceptions are:

- Card readers (input only)
- Card punches (output only)
- Printers (output only)
- Optical and magnetic character readers (input only)

Other characteristics that might influence your choice of the I/O media apply also to disk devices; they are discussed in the next section.

General Device Characteristics

Characteristics you may want to consider are:

- Data transfer rate

The number of bytes transferred within a second between external and processor storage. This rate is considerably larger for a tape or a disk compared with a card reader, for example.

- External storage capacity

  This is the amount of data that can be stored on the device; a characteristic that applies primarily to magnetic tape and disk I/O devices. In case of a disk device, part of this capacity can be used by the device itself for error checking and for synchronizing the transfer.

  A volume on a disk I/O device has to accommodate some non-data areas. Therefore, the net data capacity varies with the number and length of the records stored on the volume.

- Type of access

  Serial-only devices permit sequential read or write operations and no more. Disk devices, on the other hand, permit both sequential and direct read and write operations.

  Sequential access means reading the data one record after the other as they are stored or writing one record after the other as they are processed. Sequential access is possible on any device.

  Direct access means to find a certain record for processing it alone. On serial-only devices, this is slow because the desired record must be searched for from the beginning of the file that contains the record. On disk devices, direct access is fast because the read head of the device goes directly to the address of the desired record.

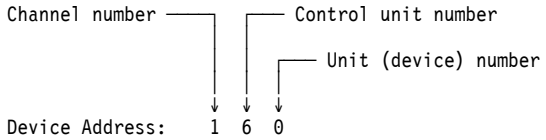
- Access time

  This applies primarily to disk devices. It is the time it takes to place the read-write head on a certain track plus the time for the desired record to rotate into the read position under the head.

- Device Addressing

  Normally, devices are attached to the system with a channel, which means that they are continually under the control of the system. Devices may also be connected over a link such as a telephone connection. However, from a device addressing point of view, there is no difference.

  When an I/O device is installed, it is attached to a channel via a separate or an integrated control unit. One or more control units may be attached to one channel; one or more I/O devices may be attached to one control unit. Each device has a hexadecimal three-digit address. This address represents hardware relationship as shown by the example below:



Disk Volume

A disk volume, also referred to as disk pack, consists of a number of disks, one above the other. Each disk, except the top and bottom ones, has two surfaces for storing data; the top and bottom disks have only one such surface.

Addressing a Disk Device

VSE supports two different addressing concepts for disk devices: the cylinder concept for CKD (count-key-data) and ECKD* (extended count-key-data) disks and the relative-block concept for FBA (fixed block architecture) disks. Whenever CKD devices are mentioned in this publication, this includes also ECKD devices.)

The Cylinder Concept

Figure 3 on page 8 illustrates this concept, which applies to CKD and ECKD disk devices.

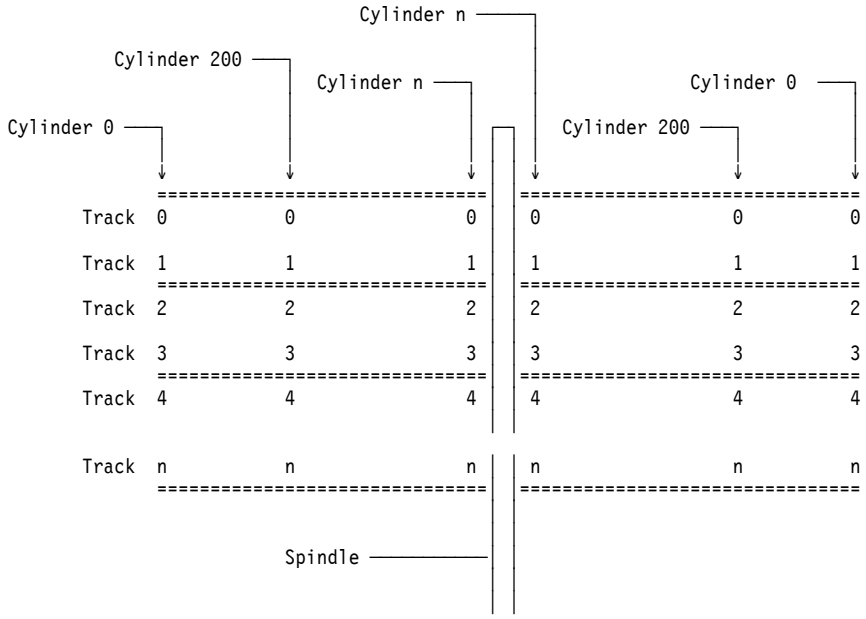


Figure 3. Cylinders and Tracks on a Volume of a CKD or ECKD Disk Device

Each disk of a volume on a CKD or ECKD disk device has a certain number of concentrically arranged tracks, and each of these tracks has the same data capacity. Tracks on all surfaces located above each other are looked at as a cylinder containing as many tracks as there are writing surfaces on the volume.

Each surface is accessed either by one read-write head that can be moved from one cylinder to another, or by a set of fixed read-write heads, one for each track.

The Relative-Block Concept

This concept applies to FBA devices. All data is stored in blocks of fixed length (corresponding to the unit of data transfer implemented for the disk device). Each block, also called control interval, is addressed by its number relative to the beginning of the volume.

Data Formats

The two addressing concepts described above require two different data formats: the count-key-data format and the fixed-block-architecture format.

Count-Key-Data (CKD or ECKD) Format

Figure 4 on page 9 shows how records are stored on tracks 0, 1, and 2 of cylinder 3 of a volume.

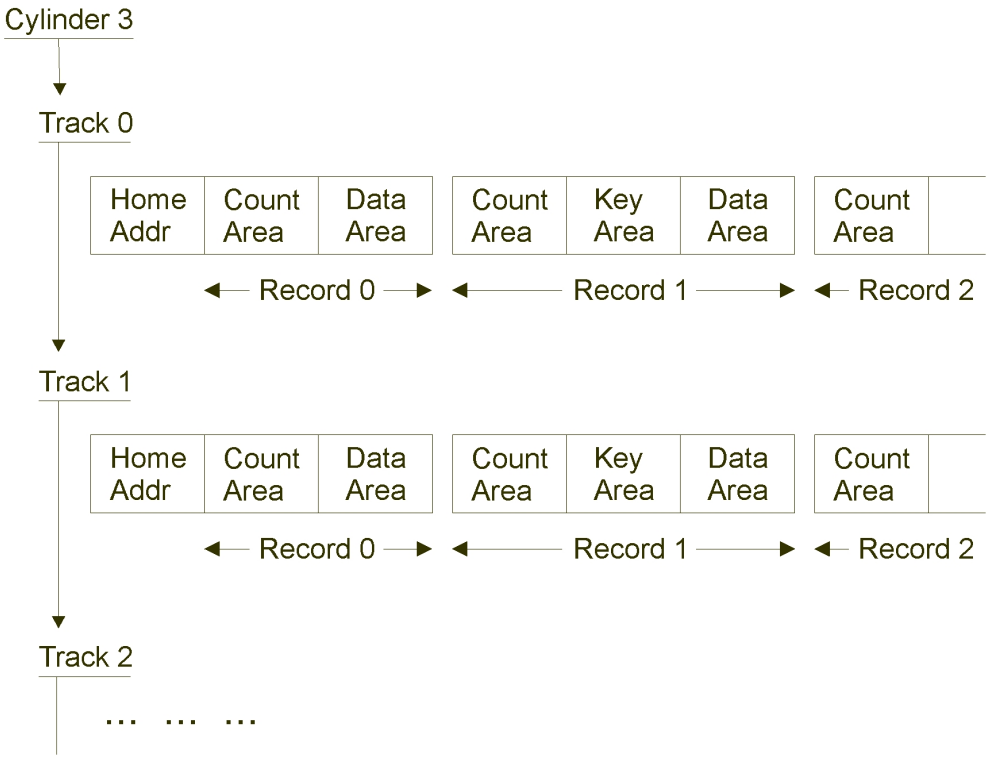


Figure 4. Track and Record Formats for CKD and ECKD Devices

Figure 4 on page 9 shows that each track has, at its beginning:

1. The track address, which is called the home address (HA). It contains:
   - The address in the form *cchh*, where:
     - *cc* = Cylinder number
     - *hh* = Head number
   - A flag byte indicating the condition of the track (such as alternate or defective).

   The home address of a track always begins immediately behind the hardware-index marker (hardware positioning) on the track.
2. A descriptor record, also referred to as record zero (R0). This record, which is always the first one on a track, is used for alternate and defective track handling. The record contains two areas:

   Count area
   Data area

The descriptor record is followed by data records. Each data record on a track contains:

A count area (to find the data area within the record)
A key area (optional, may be used to identify the record)
A data area (containing the actual data stored in the record)

Fixed-Block-Architecture (FBA) Format

For an FBA disk device, tracks and records are formatted at the time of manufacture. There is no track home address as on a CKD or ECKD device.

As Figure 5 on page 10 shows, each track contains a certain number of blocks, and each block contains:

- An ID area (containing the address of the block itself or of an alternate block if the block is defective).
- A data area (a fixed number of bytes).

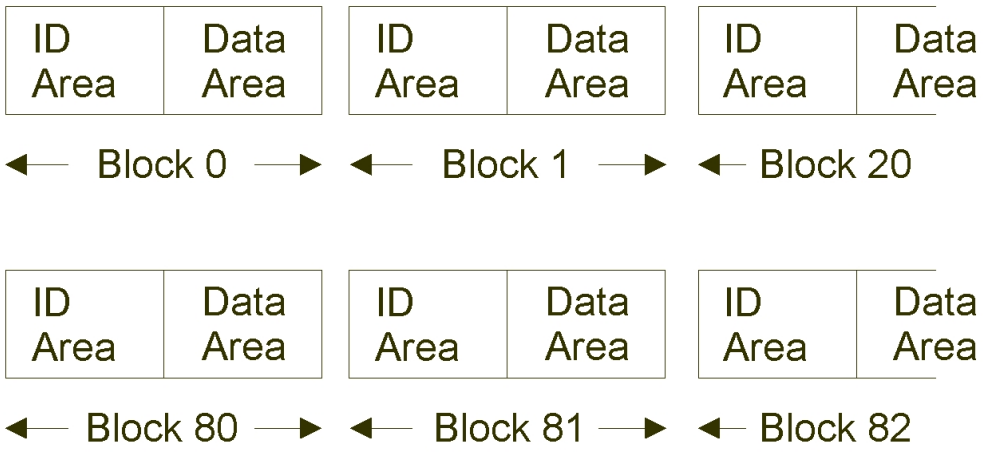


Figure 5. Blocks as Stored on a Volume of an FBA Device

Alternate Tracks or Blocks

Not all CKD tracks or FBA blocks are available for primary use; some are reserved as alternate tracks or blocks to contain data in place of a defective primary track or block.

Disk Extents

An extent is a certain area on a disk volume defined as a number of tracks or blocks and beginning at a certain track or block relative to the beginning of the volume.

A special case is a split-cylinder extent. It occupies, for example, only tracks 0 through 6 out of 10 tracks on each of the cylinders of a certain range. Split-cylinder extents can save access time if two or more related files are placed in the same range of cylinders. To retrieve related records from different files on the same cylinder, for instance, requires little or no movement of the disk drive's access mechanism. Changing tracks – switching from one read/write head to another – occurs at electronic switching speed.

You can define a split-cylinder by a job control EXTENT statement. Refer to z/VSE System Control Statements for details.

Volume Initialization

A disk volume must be initialized to contain an IBM standard volume label:

- On cylinder 0, track 0, record 3 for a CKD device.
- In block 1 for an FBA device.

To do this, use the IBM program Device Support Facilities. For more details, see the *Device Support Facilities* publication.

The volume label includes, besides the label identifier, control information as follows:

1. The volume serial number
2. The address of the volume table of contents (VTOC)

The VTOC is an area reserved on every disk volume. The area contains file labels, that is, certain information about all files residing on the volume. On a CKD disk volume, this area must be contained within one cylinder.

Tape Volume

Records stored on tape are separated by spaces called gaps. These gaps are needed to let the tape accelerate to read or write speed before reading or writing takes place. They are needed to let the tape come to a halt after reading or writing a record is finished.

On some IBM tape devices, it is possible to read or to write without any stops between records. This kind of operation is called streaming mode.

A tape can have 7 or 9 tracks. This has little effect on the use of a tape device. However, the system needs to be informed about it before the tape is being accessed. A 9-track tape volume can always be read forward or backward. Not so a 7-track volume.

Tape volumes are *initialized* with an IBM supplied utility program called INTTP. For information about using this program, see z/VSE System Utilities.

You can initialize a tape volume with an IBM or an ANSI-standard volume label written as the first record on the tape. If user-standard labels are to be written, this first record is required. Label handling routines check the volume label and, if no standard volume label is present on an output tape, prompt the operator so that he can supply a volume serial number. This then enables the label handling routines to write a volume label onto the tape. A standard volume label for tape contains:

- The label identifier (any from VOL1 through VOL8)
- The volume serial number
- User defined control information

A tape volume may be left unlabeled or it may have nonstandard labels.

This ends the discussion on hardware related device characteristics. The next section discusses how data stored on disk relates to these characteristics.

File, Extent, and Volume Relationship

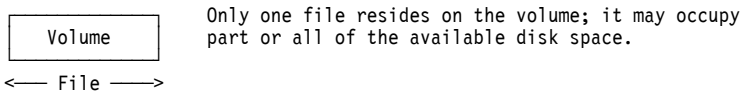
Skip this section, if you are not concerned with data on disk.

A file is a set of related records. A typical example of a file is the collection of a company's pay data – one record per employee.

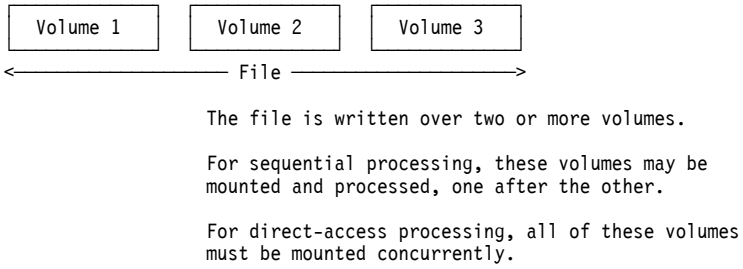
The records of a file normally have a uniform structure for use in one or more applications. Finding the most suitable structure for the records of a file is, in fact, an important tasks of developing an application program.

How a file relates to the extents on one or more volumes depends on the size of the file and the available disk space. Possible relationships are shown in Figure 6 on page 12.

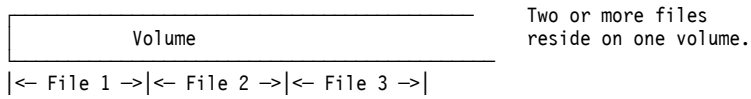
Single File/Single Volume



Multi-Volume File



Multi-File Volume



Multi-Extent File One Volume

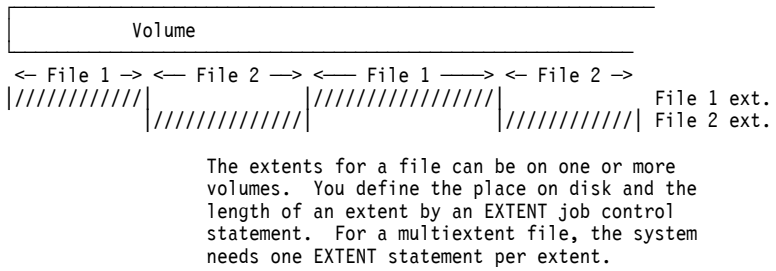


Figure 6. File, Extent, and Volume Relationships

Record, Block, and Control Interval

A file is composed of records. These records may be stored in blocks of two or more records (to save space and to reduce the number of I/O requests, for example). Each such block is, in fact, a physical record consisting of two or more logical records.

A control interval (CI), the unit of transfer for an FBA disk, may contain one or more unblocked logical records. It may contain one or more blocks of logical records; it may contain only part of a logical record.

This section describes the record formats that you can use for the various device types. It discusses control information that you may have to supply in your program.

The Records of a File

A record of a file – also referred to as a logical record – is a collection of related fields of information. For each field, you define in your program:

- The data type (binary or character, for example).
- The length to hold the largest item of data that may occur.

The sum of all field lengths in a record is the length of the record. Figure 7 on page 13 gives an example of the layout of a logical record.

Figure 7. Logical Record

Frequently, the length of a file's logical records is always the same. If this length is the same as the length of the unit of data transfer, this file is said to have fixed-length unblocked records.

A list of names, for example, can be defined as a fixed-length record file, one record per name. Most likely, the names do not have the same length. This means that names shorter than the specified length are padded with blanks and longer names are truncated to make them fit into the name field of the record.

The various record formats that you may choose for a file are discussed in this section:

Variable-Length Records

An insurance company's file of holders is a typical application that works with records of variable length. The more claims there were against a holder the longer is that holder's record in the file. Figure 8 on page 13 shows the format of a stored record in such a file of policy holders.

Figure 8. Stored Record of Variable Length

where:

BL =
Block length: A four-byte field called block descriptor. Needed also for variable-length unblocked records because they are considered as blocks with a blocking factor of 1. The value in BL includes the length of both BL and RL.

RL =
Record length: A four-byte field called record descriptor. The value in RL includes the length of field RL.

The block and record lengths are to be stored in fields BL and RL as follows:

Bytes
Contents

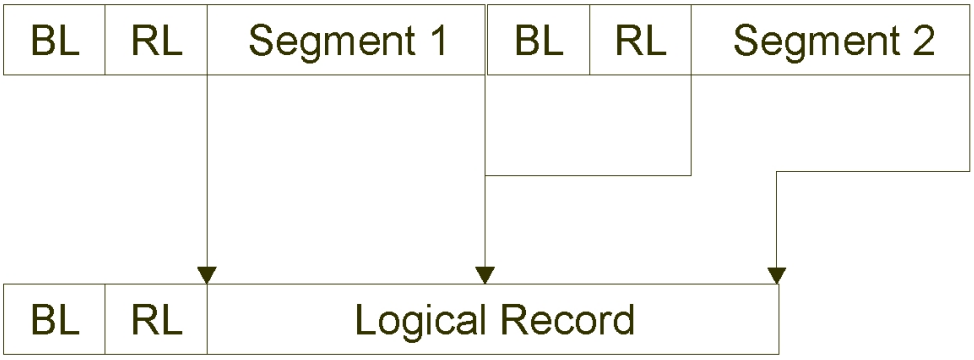
0-1
Length in binary format

2-3
Reserved.

Spanned Records

This is the record format when the records of a file are too long to fit into the given block size. One part of the record is in one block and the remainder in another. The two parts have to be reassembled again for processing. The system does this automatically for EBCDIC records.

Figure 9 on page 14 shows how records are divided into variable length segments.



BL =
Block-Length field

RL =
Record-Length field

For BL, the format description given in Figure 8 on page 13 applies.

For RL, the format is as follows:

```
  X'LLLL0f00'
where
f=0: only segment
f=1: first segment
f=2: last segment
f=3: middle segment.
```

Figure 9. Format of a Spanned Record

Spanned records may be useful when a file is to be moved between device types allowing different block sizes. The maximum block size of the receiving device may be smaller than one record. In this case every record has to be cut into segments which is done by IOCS. Another example when spanned records may be useful is in text processing applications where very long strings of text must be written. You need not be concerned about the maximum data capacity of the I/O areas. IOCS divides your records into segments that never exceed the size of the output area in your program.

Undefined Records

Any record format that does not conform to the rules for fixed- or variable-length records is considered an undefined record. IOCS allows a program to process such records but does not support this processing. Therefore, if you want to block or deblock such records, your program must provide for these functions.

Programs that write undefined records must communicate the size of each record to IOCS. Programs that read such records are informed of their length by IOCS routines.

Block of Records

To save time and space in processing, records can be grouped into blocks. This results in larger transfer units. For example, data stored on tape by one write operation is separated from the data stored by the next write operation by an inter-record gap. The smaller your records are, the more gaps (unused space) occur within a file of data. Gaps allow the tape device to accelerate before it starts to read or write the data. They allow the tape to come to a halt after having read or written a record of data.

The time needed to start and stop a tape is significant for the overall speed with which your program can read from or write to a tape. Fewer gaps therefore result in faster processing.

To reduce the number of gaps, you can group two or more records into a block, a technique called blocking. The number of records in one block is called blocking factor. Figure 10 on page 15 illustrates the blocking of records.

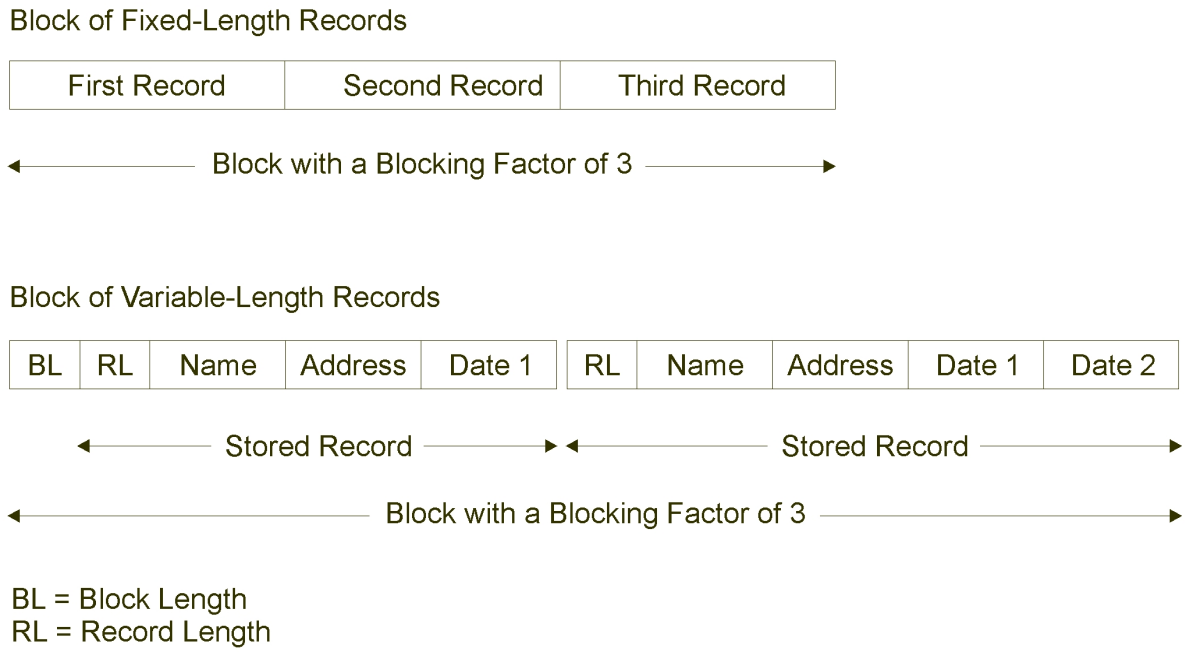


Figure 10. Block and Blocking Factor

Control Interval

The unit of transfer for an FBA disk is the control interval (CI). Its smallest size is 512 bytes, the size of an FBA block; but you can choose to define a CI size of two or more FBA blocks, if this is desirable. However, since a CI on an FBA disk is always addressed by its relative FBA-block number, the CI size must be a multiple of 512. Normally, one FBA block is also one CI.

A CI can hold one or more logical records. It can include free space.

CI format is used also by VSE/VSAM, an access method which is part of z/VSE and discussed in separate publications. Control-interval format in this context therefore refers to sequentially organized files stored on an FBA disk.

Figure 11 on page 16 shows one control interval comprised of two FBA blocks. It contains:

- Two data blocks
- Free space
- Control information for each of the data blocks (RDFs)
- Control information for the CI (CIDF)

You can optimize space utilization on an FBA disk by choosing a blocking factor that leaves little or no free space in a CI.

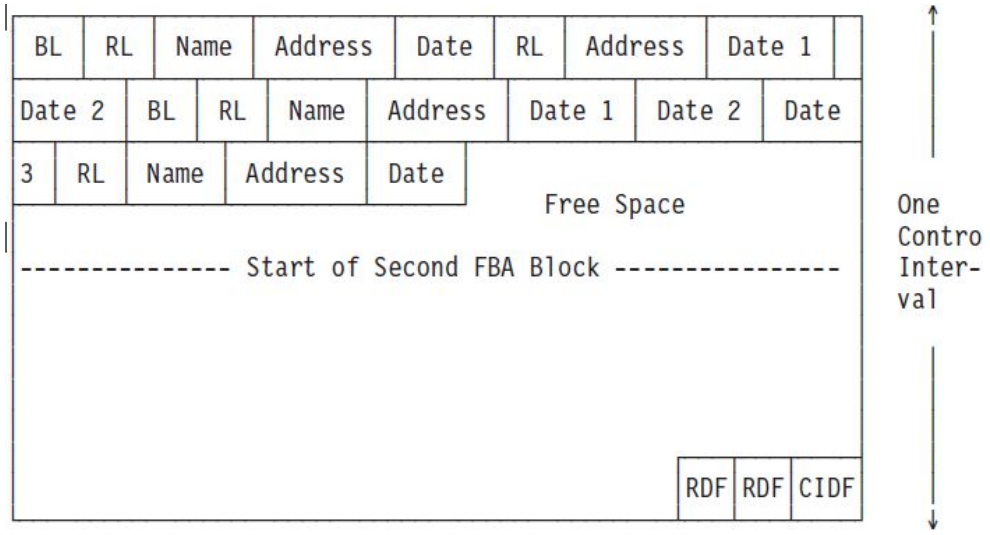


Figure 11. Layout of a Control Interval

Blocking and Deblocking of Logical Records

Once you have defined your blocking factor in your program, the IOCS routines automatically handle blocking and deblocking of logical records. An I/O request (such as a GET or a PUT macro) in your program causes the next record of a block to be made available to your program on input or to be added to the current block on output. This applies also to I/O from and to an FBA disk. Your program does not become aware that the unit of transfer is a CI rather than a block. The IOCS routines handle the data transfer between the device and the I/O areas in your program.

Device-Dependent Record Formats

Although the IOCS routines ensure that all hardware requirements are met, these formats are briefly discussed. You may need to know about these formats if you have to design or implement a complex application system.

Records on a CKD or ECKD Disk

A physical record on a CKD or ECKD disk consists of three distinct areas:

- Count area
- Key area (optional)
- Data area

Figure 12 on page 17 shows the format of a record with and without key area.

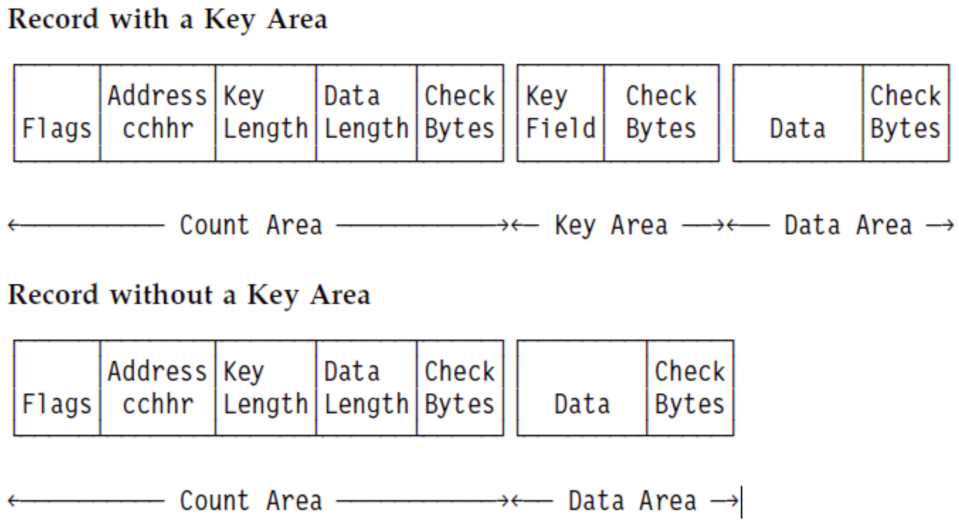


Figure 12. Hardware-Dependent Format of a CKD or ECKD Recordl

Areas of interest are briefly discussed below:

- Count area

  This area is recorded automatically and maintained by the system. The address field contains the address of the data, where:

  cc =
  Cylinder number

  hh =
  Head (track) number

  r =
  Record number

  The record number is the sequential position of the record on the track.

- Optional key area

  The key field of this area contains an identifier of the record, such as a part number. If the data is comprised of a block of logical records, this identifier is the highest or lowest key of these records in the block. The field can have a length of up to 255 bytes.

- Data area

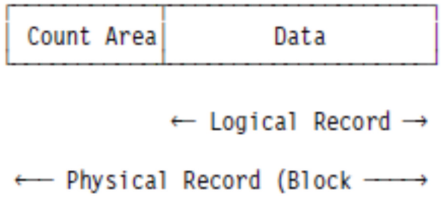
  The area contains the data, which may be just one logical record, a number of records, or only a part (segment) of a logical record if this is longer than the unit of transfer. A record of that length, a spanned record, must have its key in the first segment if the key area is not used. A record on disk cannot span volumes.

- Check bytes

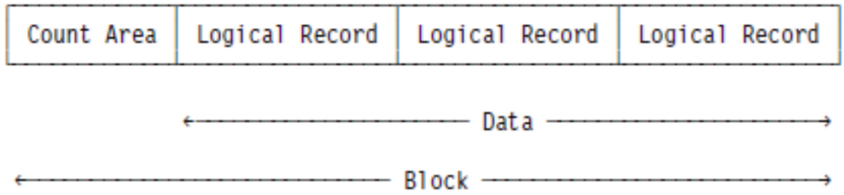
  They are maintained by the system.

Figure 13 on page 18 shows the format of the data area for records of fixed and variable length format, unblocked and blocked. Figure 14 on page 19 shows an example of spanned records on a CKD or ECKD device.

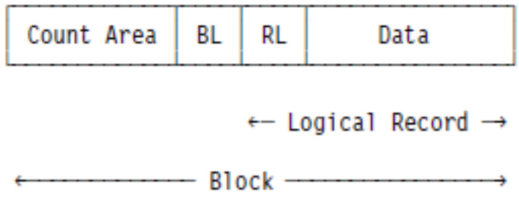
Fixed-Length Unblocked CKD or ECKD Records



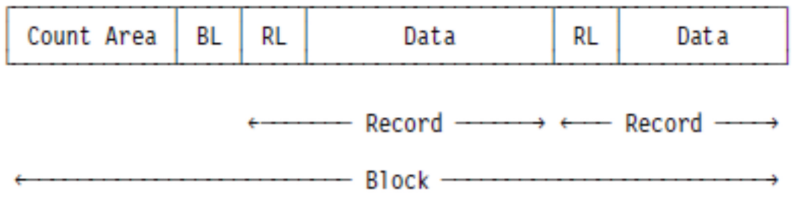
Fixed-Length Blocked CKD or ECKD Records (a blocking factor of 3)



Variable-Length Unblocked Records



Variable-Length Blocked Records (a blocking factor of 2)

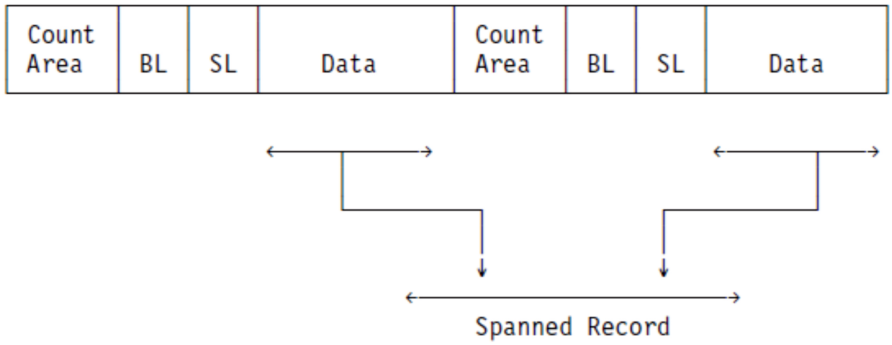


BL = Block Length Field (does not include the length of the count area)

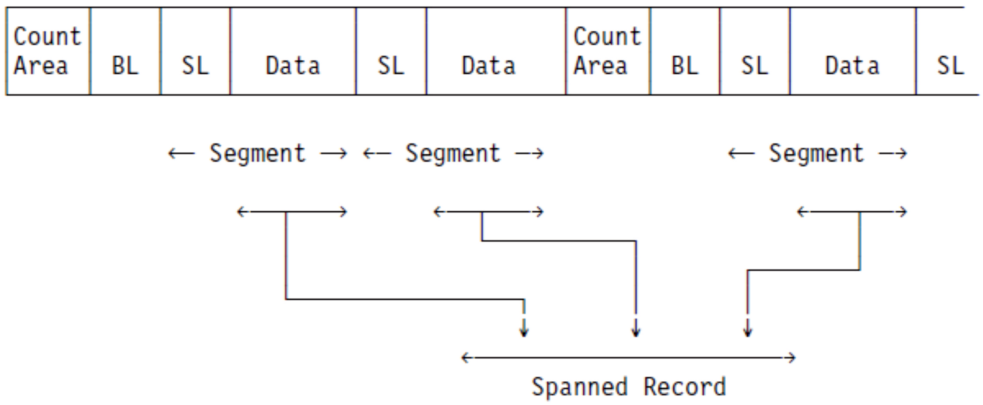
RL = Record Length Field

Figure 13. Records of Fixed and Variable Length (Without Key Area) on a CKD Disk

Unblocked Spanned CKD or ECKD Records



Blocked Spanned CKD or ECKD Records



BL =
Block Length Field

SL =
Segment Descriptor Field: Gives the length of the segment and tells whether the segment is the first, only, last, or middle one of the record.

Figure 14. Spanned Records on CKD or ECKD Devices

Records on a FBA Disk

Figure 11 on page 16 shows how data may be stored in a CI. It shows where in a CI the control information (one or more RDFs and a CDIF) is stored.

A record or block of records can be shorter or longer than an FBA block. In other words, a record or block may span CI boundaries. If you define this to your system's IOCS routines, IOCS automatically:

- Assembles the complete record in your program's input area on input.
- Writes the complete record to disk as segments that fit the CIs as defined in your program on output.

Records on Magnetic Tape

All standard record formats, including ASCII-coded records, are acceptable for magnetic tape. Spanned records (EBCDIC only) may span volumes.

The records or blocks of an ASCII coded file may be preceded by a block prefix which can be up to 99 bytes long. However, ASCII records are processed in EBCDIC, and the system takes care of the conversion and translation.

Records for a Printer

Printed output is always unblocked. The length of a record depends on the maximum length of a print line.

Carriage control for a printer can be specified to a certain extent by a control character in the data records.

For an IBM 3800, you can select a character arrangement table by specifying a reference number preceding the control character for carriage control.

Most printers have control buffers to control operations such as spacing over more than three lines (also referred to as skips of forms), page eject, or the character set to be used. These buffers have to be loaded with buffer images that meet the needs of your program. The buffers are:

- Forms-control buffer (FCB)

  The buffer describes the layout of the printed page. It determines, for example, where the first print line is placed on the page, how many lines per inch have to be printed, and where the last line on the page is to be.
- Universal character-set buffer (UCB)

  The buffer describes the character set of the mounted print band (or train).

The loading of a control buffer cannot be triggered by control information included in a print record. However, since the contents of these buffers affect the format of your output page, loading the buffers is briefly discussed below.

A print-control buffer can be loaded:

- Automatically by IPL during system startup.
- By the operator when issuing the LFCB or LUCB command.
- From within your program by coding the LFCB macro.
- As a separate job step by executing the SYSBUFLD program. For information about the operator commands and about using the SYSBUFLD program, see z/VSE System Control Statements.
- Under VSE/POWER via the $$ LST statement. The statement is described in VSE/POWER Administration and Operation.

Records on a Card Device

Card records for input are always unblocked and have a length of up to 80 or 96 characters, depending on the characteristics of the card reader.

Card output may be any unblocked format. When variable-length records are punched, the length fields for the records (BL and RL) are not punched.

You can specify the stacker that has to be used for output by a control character in the data records.

Records on the Console

Records may be entered or displayed in fixed-length or undefined format. The length of a record may not exceed 256 characters.

Organization of Records in a File

The records in a file are sequenced, either as they were entered or sorted by one of their fields (containing a key) or by a key external to their data content. Following is a discussion of the most commonly used ways of organizing the records of a file:

Without Keys for Sequential Access

The records of the file are read (on input) or written (on output) as they are presented. A typical example of such a file is the lines of data written to a printer.

To find a certain record of the file, your program must request successive read operations until this record is retrieved from the file.

With Keys for Sequential and Direct Access

This applies only to a file on disk. IBM's VSE/VSAM is the access method that fully supports this way of organizing the records of a file.

The records of the file can be sorted in ascending order of their keys. A number for each record, for example, or any other data of significance. The file might look as shown in Figure 15 on page 21.

Sorted by Personnel Number

```
0015   John Jones      5A    23400    650102
0059   Ken Baker       88B   12000    870302
0073   Stan Kowalski   3C    16200    800601
0369   Jacob Ferrari   2D    24600    710301
0572   Peter Smith     5A    21000    700901
0573   Jerry Tames     65D   30000    601001
0773   Jim Brown       3C    18000    751115
0928   Mike Rose       4B    18000    851115
1457   Tom Hatfield    50B   26400    650102
```

Sorted by Department Number

```
0369   Jacob Ferrari   2D    24600    710301
0073   Stan Kowalski   3C    16200    800601
0773   Jim Brown       3C    18000    751115
0928   Mike Rose       4B    18000    851115
0015   John Jones      5A    23400    650102
0572   Peter Smith     5A    21000    700901
1457   Tom Hatfield    50B   26400    650102
0573   Jerry Tames     65D   30000    601001
0059   Ken Baker       88B   12000    870302
```

Figure 15. The Records of a File with Keys

If the keys are not unique (if there were persons with the same name, for instance), a secondary key field can help. This method of using such a key organization is comparable to the use of a telephone directory. It allows for one or more indexes to be built for the file to speed up direct access. Such an index would include the key of every record in the file together with the record's address.

As “Organization of Records in a File” on page 20 shows, the records of a file with an index need not be ordered in key sequence. Only the index must be ordered in that sequence. By locating the key of the desired record, the access method can retrieve the record directly.

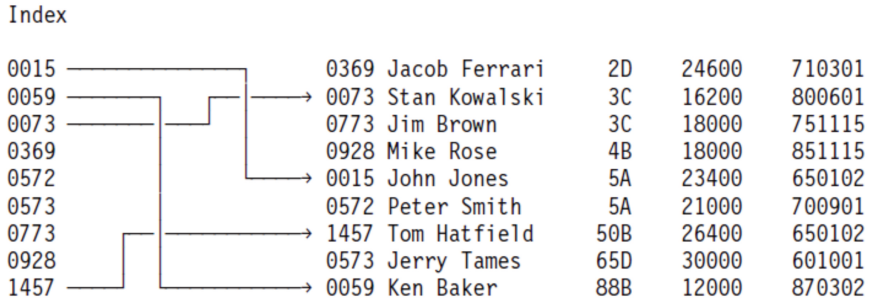


Figure 16. Locating a Record Via an Index

Choosing the Right Access Method

For you as an application programmer, an access method is a set of macros that:

1. Define the characteristics of the data that your program is to process.
2. Request an I/O service at those points in your program where this service is required.

An access method provides you with the functions needed to store and to retrieve your data. Based on your definitions, the access method links your program with the selected I/O device.

Following are some of the major criteria for choosing the right access method:

- The mode of processing (sequential or direct access)
- Scope of support by the access method
- Available programming language support

Sequential Versus Direct Processing

The efficiency of sequential processing depends on the percentage of records handled in one run and on the size of your data blocks. If a file is used frequently or if the transactions for single records can be saved up for some time, a high percentage of the records can be handled in one run. Sequential processing is indicated in this case. For sparse activity, or if immediate processing is required, direct processing is the preferred method.

The following rule of thumb may help you in making the right decision: if the number of blocks is smaller than the number of affected records multiplied by 1.7, sequential processing is faster.

Level of Support

Quite frequently, the level of IOCS service offered by an access method is a significant selection criterion.

Table 1 on page 22 might help you in making an initial selection. A more detailed summary of the support by the available access methods follows Table 1 on page 22.

Table 1. Available Access Methods and their Scope of Support

Access Method	Scope of Support	Supported Device Classes
Sequential Access Method (SAM)	Simple, sequential applications. No automatic extensions of the file. No space management.	All
SAM for a file in VSAM-managed space	Same as above, but automatic extensions of the file are possible.	Disk
VSE/VSAM (see Note)	Applications that require sequential access, direct access by key, or both. Automatic extension of a file is possible. Includes routines for managing the available disk space.	Disk
Direct Access Method (DAM)	Inquiry type applications. Access is by key or by record address as determined by a randomizing algorithm in your program.	CKD or ECKD disks only
Indexed-Sequential Access Method (ISAM)	An access method for which IBM has discontinued further development. Applications that require sequential access or direct access by key or both. Automatic extension of a file is not possible. Does not include routines for space management.	Only CKD disks of earlier design: 2311 2314/2319 3330/3333 3340/3344

Note: This is the recommended access method if the records of a file are to be processed both sequentially and direct. For more information about VSE/VSAM, see the publications:

- VSE/VSAM Commands
- VSE/VSAM User's Guide and Application Programming

Sequential Access Method (SAM)

You can use SAM for the processing of files on all supported I/O devices. It stores and retrieves the records of a file, one record after the other. It can process these records blocked or unblocked.

If the accessed file resides on disk, SAM can update a file in place.

SAM is efficient if most or all records are to be processed in sequential order. The access method is discussed in more detail in the subsequent topics of this publication.

VSE/VSAM

An access method available as an IBM licensed program. It is one of the base programs of z/VSE.

This access method offers automatic space allocation and supports alternate indexes. This makes your operation less dependent on device characteristics. The access method manages the available disk space and the files stored in this space. It uses for that purpose a master catalog and, optionally, one or more user catalogs. VSE/VSAM includes a service program, Access Method Services, which can:

- Define and maintain a VSAM file
- Load records into a VSAM file
- Build one or more alternate indexes for a VSAM file
- Copy and print a file
- Create a backup copy of a VSAM file and restore this copy
- Recover from damage to data
- Convert a SAM or ISAM file to the VSAM format

VSE/VSAM allows you to choose from four different file (data set) organizations:

- Key-sequenced data set (KSDS)
- Entry-sequenced data set (ESDS)
- Relative-record data set (RRDS)
- Variable-length relative-record data set (VRDS)

All of which offer additional processing options. VSE/VSAM supports, in addition, SAM ESDS files to be accessed with SAM macros. For details about this support, see the VSE/VSAM User's Guide and Application Programming.

Using VSE/VSAM is not discussed in this publication. For more information about the access method, see:

- VSE/VSAM Commands
- VSE/VSAM User's Guide and Application Programming.

Direct Access Method (DAM)

DAM handles direct-access processing of a file on CKD disk devices only. It supports all unblocked record formats.

The records of the file can have fixed-length keys but need not be sorted by those keys. Your program, however, must include an algorithm that converts a key to a valid disk address. If the records of a file do not have keys, they are identified by their track addresses.

DAM *does not* support any of the following functions; your program must provide for them:

- Handle overflow records
- Locate synonym records
- Delete records
- Process a file that is not entirely online

A DAM file must be online (that is, all of its volumes must be mounted) whenever it is processed. To reorganize the file, either disk space of twice the size of the file must be online or you must use a tape as intermediate storage.

For more information about this access method, turn to Appendix B, “Direct Access Method (DAM),” on page 189.

Indexed Sequential Access Method (ISAM)

ISAM is an access method that supports sequential and also direct processing of data on CKD disk devices of earlier design.

To process a file, ISAM requires that the file's records have keys of a fixed length. ISAM maintains a two- or three-level index of the highest record keys on the tracks used by a file.

ISAM can handle a file overflow within the limits of available extents.

ISAM *does not*:

- Handle records of variable length
- Handle a file that is not entirely online

For performance reasons, a file accessed by ISAM needs to be reorganized from time to time. This requires that twice the size of the file is kept online or that a tape is used as intermediate storage.

IBM has discontinued further development effort for this access method. Consider using VSE/VSAM instead of ISAM if certain data on disk is to be accessed both sequentially and direct.

The access method is not discussed any further in this publication. The macros in support of this access method are documented in z/VSE System Macros Reference.

Support of Access Methods by Programming Languages

An overview of the access methods supported in the various programming languages is given in Table 2 on page 24.

Table 2. Access Methods

Programming Language	Supported Access Method			
	SAM	**VSAM**	**DAM**	**ISAM**
COBOL	YES	YES	YES	YES
C/370	YES	YES	YES	YES
VS FORTRAN	YES	YES*	YES**	NO
RPG II*	YES	YES	YES	YES
PL/I	YES	YES	YES***	YES
Assembler	YES	YES	YES	YES

*****
Only ESDS and RRDS.

******
The user specifies the relative record numbers in the file.

****
With keys or with user-specified relative record numbers.

This ends the introduction to IOCS under VSE. The next topic discusses the kinds of labels that are stored on disk or tape volumes.

Volume and File Labels

Volume and file labels are records stored on a storage medium such as a disk or a tape volume. Their main purpose is to electronically identify the volume on which they are stored and the file with which they are associated. VSE can process labels that follow the formats summarized below:

- IBM Standard Labels

  Records of a fixed length. IBM standard labels are required for disk volumes and for files on these volumes. They are optional for tape volumes.

- User-Standard Labels

  Records of a fixed length. They apply to files on disk or on tape, and they are meaningful for files whose records are processed sequentially.

- Nonstandard Labels

  These labels do not conform to the specifications for IBM or user-standard labels. They may have any length and may contain any information that you desire. Programming for the processing of nonstandard labels can be quite cumbersome, however.

VSE supports these labels as follows:

- For a volume

  The IBM standard volume label – A volume that has standard labels must have a volume label. The label has the identifier VOL1 in its first four character positions.

- For a file

  IBM standard labels – If a volume has a standard label, every file residing on the volume must have at least one file label. An IBM standard file label on disk contains the name of the associated file. An IBM standard file label on tape contains the following in its first four character positions:

  – HDRn = For header label n
  – EOFn = For end-of-file label n
  – EOVn = For end-of-volume label n

  User-standard labels – If a volume has IBM standard labels, a file may also have user-standard labels. A user-standard file label has the following identifier in its first four character positions:

  – UHLn = For user-header label n
  – UTLn = For user-trailer label n

"Volume and File Labels" on page 25 shows which labels are mandatory (M) for the various types of devices and which labels are optional (O).

Table 3. Volume and File Labels as Supported by a VSE System

	Label Identifier		
	Volume Label	Standard File Label	
		IBM	User
Disk **Sequential (DTFSD) and Direct (DTFDA) only**	VOL 1 M	File Identifier M	UHLn O UTLn O
Tape **EBCDIC and ASCI**	VOL 1 O	HDR1, HDR2 O EOF1, EOF2 EOV1, EOV2	UHLn O UTLn O

For tape volumes and files, VSE allows you to use nonstandard labels or no labels.

IOCS writes the required labels for an output file, based on information that you supply in one of the following ways:

- In the DTFxx macro that defines the file.
- By job control label information (DLBL, EXTENT, or TLBL statements)
- By both.

For a description of the DLBL, EXTENT and TLBL statements, see z/VSE System Control Statements. How to use these statements is discussed in z/VSE Guide to System Functions.

IOCS checks the existing labels for an input file to verify that the correct volume and file is being accessed.

Volume Labels

A volume label (VOL1) is written onto a disk or tape volume when the volume is initialized.

The initialization programs write the VOL1 label at a fixed location on the volume.

Initializing a Disk Volume

A new disk volume must be identified (labeled) on the system that will use the volume. To do this, you use the Device Support Facilities program of your VSE system.

You can have the program write up to seven additional volume labels (VOL2 through VOL8). However, only a VOL1 label is supported by VSE.

The Device Support Facilities program also writes the file label for the volume table of contents (VTOC) and formats the VTOC. The VTOC is an area pointed to by the VOL1 label. This area contains the file labels of all files on the volume.

Initializing a Tape Volume

A new tape volume, if it is to be labeled, must be initialized by an Initialize Tape (INTTP) utility run. The utility writes, onto the volume, a VOL1 label and, optionally, up to seven additional volume labels (VOL2 through VOL8). However, only a VOL1 label is supported by VSE. How to use the utility is described in z/VSE System Utilities.

File Labels

Files must have labels when they are stored on disk. They can be without labels if they are stored on an unlabeled tape.

Disk File Labels

A file on disk must have at least one file label. The labels of a file identify each extent of this file on its volume. As mentioned earlier, disk file labels are stored in the volume's VTOC.

Tape File Labels

For a file on a labeled tape volume, IOCS writes a HDR1 label as the first record of the file and a HDR2 label as the second record. It writes an EOF1 and an EOF2 label at the end of the file and an EOV1 and an EOV2 label at the end of the volume.

Volume Organization

This section describes how the various types of data volumes are organized.

Disk Volume

Figure 17 on page 27 shows how the volume label and the VTOC relate to each other. It shows how IOCS uses the labels stored in the VTOC to locate the extents of the associated file.

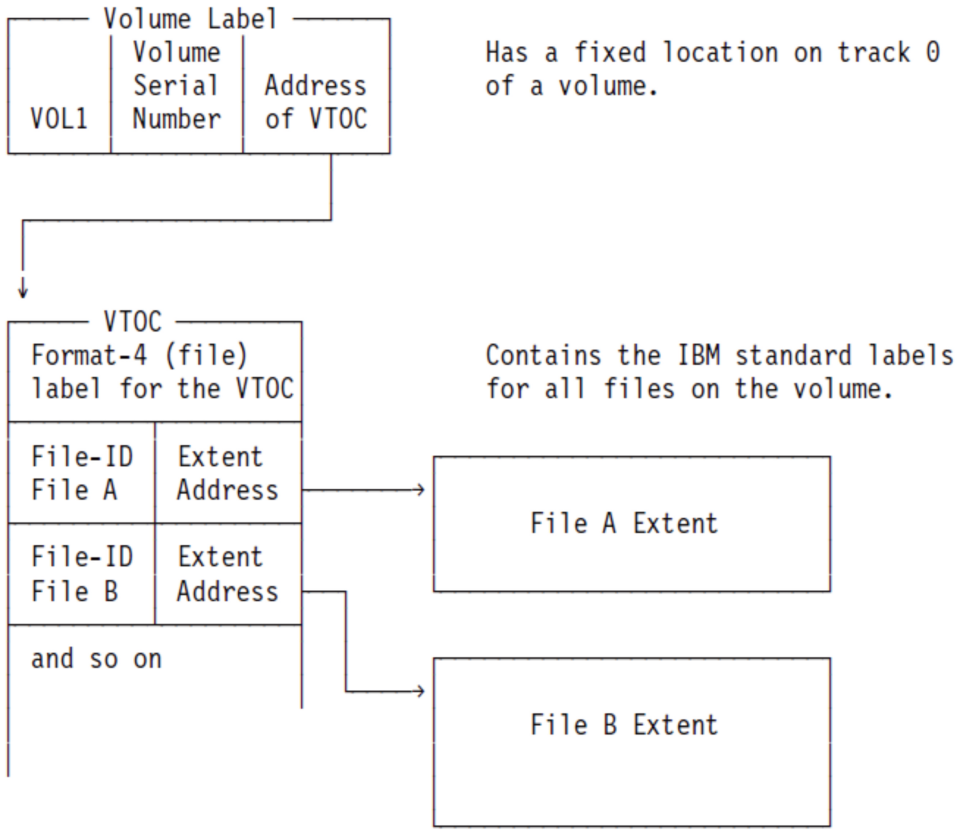


Figure 17. Disk Volume Organization – Concepts

Placement of IBM Standard Labels

On the volume of a CKD disk, the volume label always starts at record location 3 of track 0 on cylinder 0 (the first two records on this track are reserved).

On the volume of an FBA disk, the volume label is the first record in the second FBA block.

Layout of the VTOC

Figure 18 on page 28 shows the general layout of a disk VTOC with a VTOC label. Following is a summary of the purpose of the various types of labels that may be stored in a VTOC:

Format-1:
Defines up to three extents of a file or a VSAM data space.

Format-2:
Used by ISAM.

Format-3:
Needed when a file (or a VSAM data space) occupies more than 3 extents on the volume.

Format-4:
Defines the VTOC itself.

Format-5:
Not used on a VSE system.

For details about the contents of the various types of disk file labels, see z/VSE System Macros Reference.

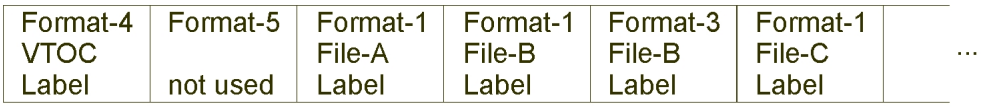


Figure 18. General VTOC Format for a Disk Volume

The Format-1 Label

Figure 19 on page 28 expands on Figure 18 on page 28 by showing the most significant fields of the format-1 label. Up to three data extents of a file are identified and located with the format-1 label of a file. One or more format-3 labels are written by IOCS if a file has more than 3 extents.

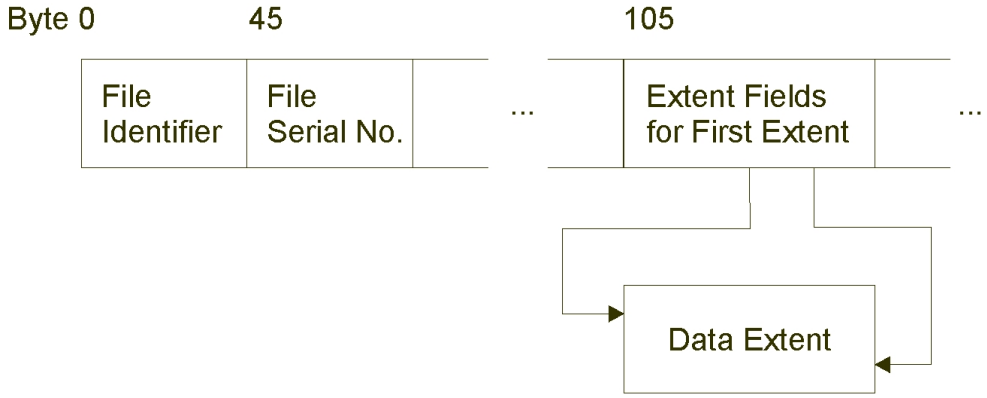


Figure 19. Format-1 Label Overview

Placement and Capacity of the VTOC

You define the location of the VTOC and its length to the system when you initialize the volume. This location cannot be within an area that is reserved for alternate use.

For an **FBA disk**, the VTOC can begin at any block location, except blocks 0 and 1 on a non-SYSRES volume. It can start at any block location behind the system area on a SYSRES volume.

Each label in the VTOC requires a three-byte record definition field. In addition, a one-byte definition field is required per control interval (CI). Therefore, to find out how many labels (L) fit into a control interval, use the formula:

$$L = (C_{CI\text{-}size} - 4) / 143 \times N_{No.\ of\ CIs}$$

The VTOC can contain up to 999 file labels.

For a **CKD disk**, the VTOC can begin on any track except track 0 on cylinder 0 on a non-SYSRES volume. It can begin at any track behind the system area on a SYSRES volume.

Note: If the prime data area of an ISAM file spans several volumes, the VTOC for the first volume must precede the prime data area. On the remaining volumes except the last, the VTOC must be on cylinder 0. On the last volume, the VTOC may be on cylinder 0 or it may follow the prime data area.

The capacity of a VTOC depends on the cylinder capacity of the disk device that is being used. If you have many small files on a volume and you can foresee VTOC capacity problems, consider placing your files under control of a VSAM catalog.

Placement of User-Standard File Labels

IOCS writes user-standard file labels on the first track of the first extent of the file on a CKD disk or in the first CI of the file on an FBA disk. Therefore, if user-standard labels are used on a CKD disk, the first extent

on the disk volume must be at least two tracks long. Your data records then start with the second track of the extent.

Disk Volume Layout Examples

The examples show in what arrangement IOCS writes labels on volumes, cylinders, and tracks on output. IOCS expects labels to be so arranged on input. Each of the examples shows:

- How the space on the volume is used (cylinder distribution).
- The layout of the track with volume labels and VTOC.

Some of the examples give the contents of extent fields of the first format-1 label and the format-4 label in the VTOC.

The examples are:

- A CKD volume of 410 cylinders with the VTOC beginning on track 1 of cylinder 100 – Figure 20 on page 29.
- A CKD volume of 202 cylinders with disk files that have user-standard labels – Figure 21 on page 30.
- A multivolume VSAM layout showing the relationship between the labels, the catalog, the data space, and the files – Figure 23 on page 32.

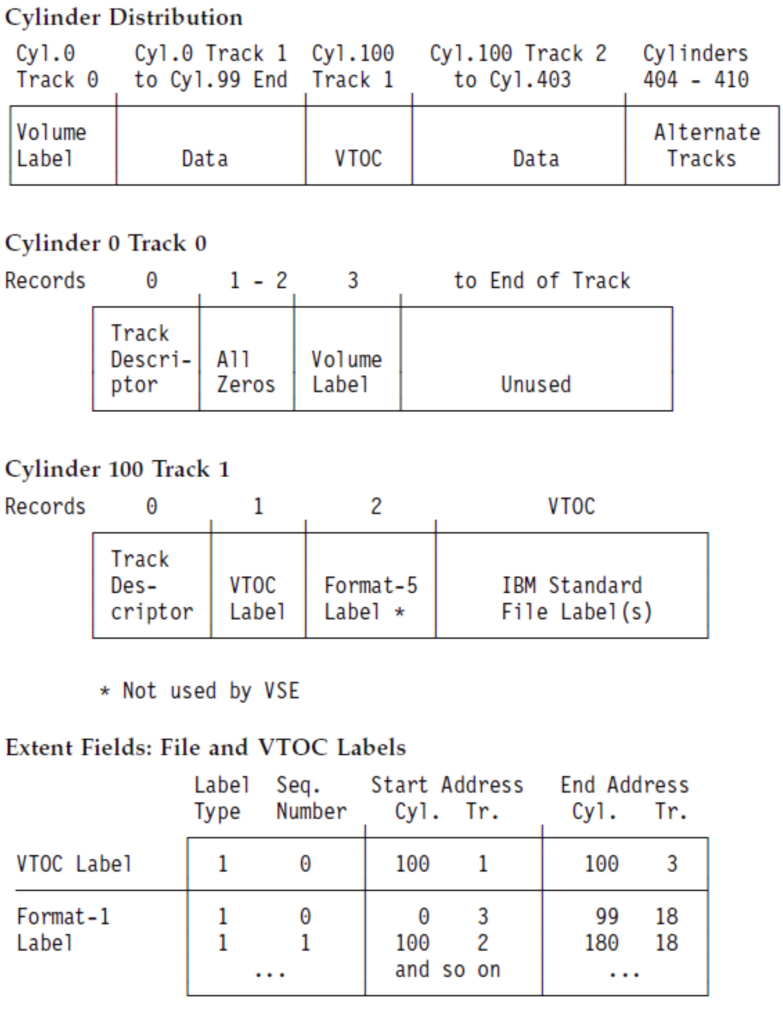


Figure 20. Disk Volume Layout: One File plus VTOC

VTOC on Cylinder 199

Volume 1

Track Des-criptor	VTOC Label	Format-5 Label *	IBM Standard File Labels		
			File A	File B	...

Volume 2

Track Des-criptor	VTOC Label	Format-5 Label *	IBM Standard File Labels		
			File A	File C	...

Volume Layout

Volume 1

VTOC	Data Space 1	VSAM Catalog	Data Space 1	Data Space 2
	File A File B		Free Space	File C *

Volume 2

VTOC	Data Space 3	SAM File	DAM File	Data Space 3
	File D			File E

VTOC Layout

Volume 1

VTOC Label	Format-5 Label **	IBM Standard File Labels for:		
		Catalog	Data Space 1	Data Space 2

Volume 2

VTOC Label	Format-5 Label **	IBM Standard File Labels for:		
		Data Space 3	DAM File	SAM File

* A unique file.

** Not used by VSE.

Figure 21. Disk Volume Layout: Files with User-Standard Labels - Part 1

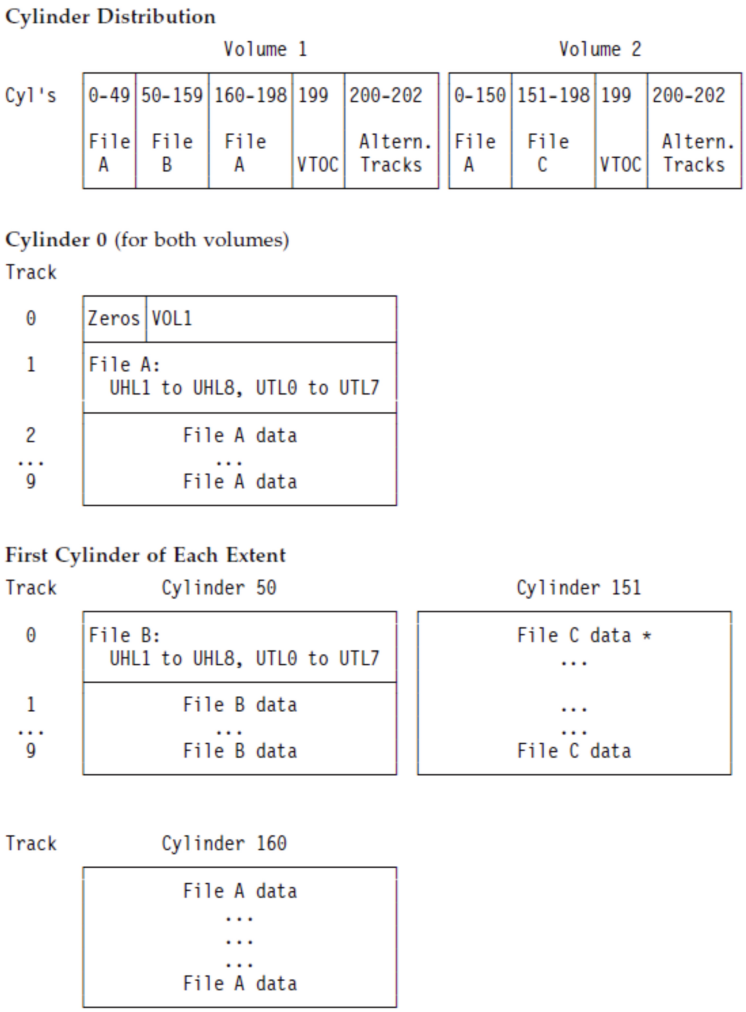
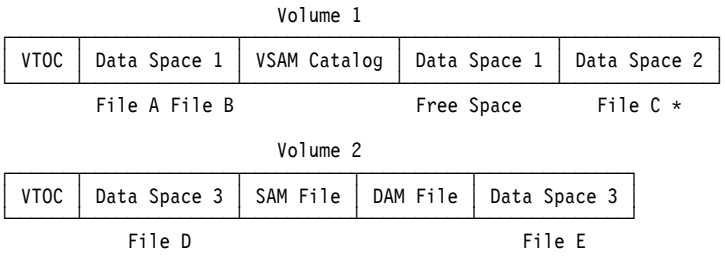
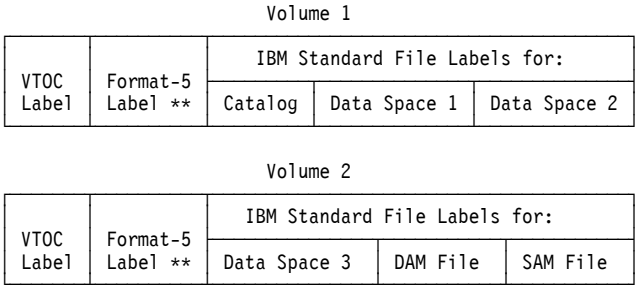


Figure 22. Disk Volume Layout: Files with User-Standard Labels - Part 2

Volume Layout



VTOC Layout



*****

A unique file.

******

Not used by VSE.

Figure 23. Disk Volume Layout with VSAM Data Spaces

Tape Volume

On a labeled tape, the first record must be a volume label (VOL1).

IBM standard tape file labels are located immediately before and after the file. A header label precedes each file and a trailer label follows each file. User-standard labels, if present, always follow the IBM standard header and trailer labels. No user-standard label can be written on a volume without IBM standard labels.

Each label is followed by an interblock gap; a tapemark separates a set of labels from the data records. Two tapemarks follow the end of the last file on a volume; one tapemark follows the last (or only) trailer label of a file.

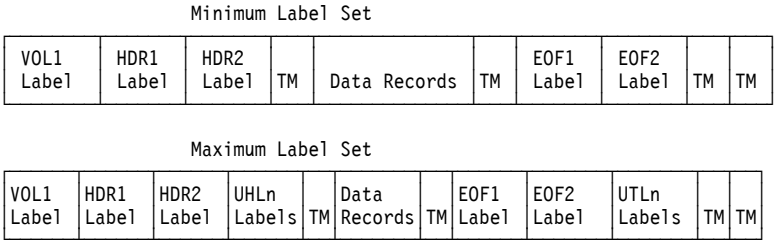
Logical IOCS writes these tapemarks automatically, except when you request LIOCS not to write them. You can do this by defining certain options in the DTFMT macro for your tape file.

Figure 24 on page 33 shows the layout of a tape volume with IBM and user-standard labels. Figure 25 on page 33 shows the layout of a volume with nonstandard labels. These layouts assume that IOCS is instructed not to suppress the writing of tapemarks.

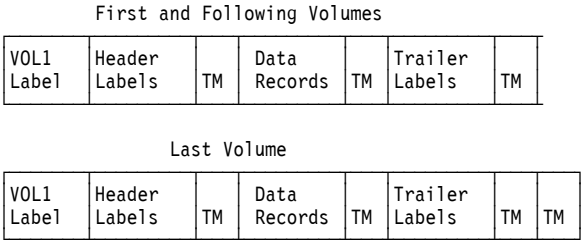
Figure 26 on page 34 shows the layout of a tape volume without labels, again with IOCS instructed not to suppress the writing of tapemarks.

Legend: TM = Tapemark

Single-Volume File



Multi-Volume File



Multi-File Volume

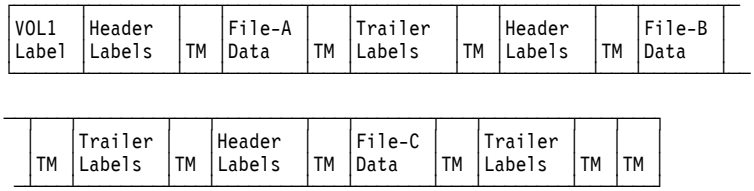
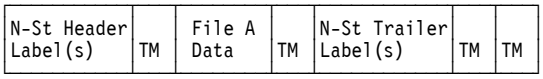


Figure 24. Tape Volumes with IBM and User-Standard Labels

Legend: TM = Tapemark; N-St = Nonstandard

Single-File Volume



Multi-File Volume

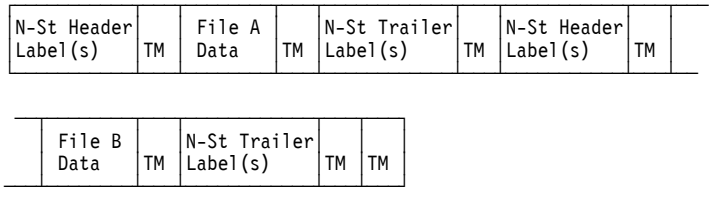
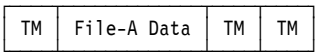


Figure 25. Tape Volumes with Nonstandard Labels

Legend: TM = Tapemark

Single-File Volume



Multi-File Volume

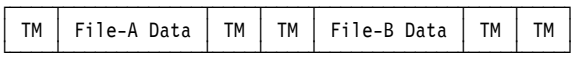


Figure 26. Tape Volume with Unlabeled Files

The Input/Output Control System (IOCS)

For you as an application programmer, IOCS is a set of macros that you can use to declare the characteristics of data and request an I/O operation. IOCS macros are divided into IOCS declarative and IOCS request macros.

- Declarative macros

  There are two related types: DTFxx (Define the file) and xxMOD (module generation).

  A DTFxx macro defines a file for a certain access method and the type of the I/O device that is to be used.

  An xxMOD macro defines the logic module that controls the execution of the I/O operation requested by a request macro.
- IOCS request macros

  An IOCS request macro identifies the I/O operation that is to be performed. Request macros are discussed throughout the publication. The GET macro for example, indicates that your program needs a record to be read into its input area.

  An IOCS request macro normally initiates the action to be performed by branching to a logic module. Linkage between your program and a logic module is established as follows:

  1. By the assembler when it assembles a DTF (table), based on your DTFxx macro and, if necessary, an xxMOD macro.
  2. By the linkage editor when it processes the assembled module(s).

Figure 27 on page 34 shows the relationship between a program, the assembled DTF, and the related logic module. The file which is processed in this example is named CARD, the name of the related logic module is IJCFAOZ0.

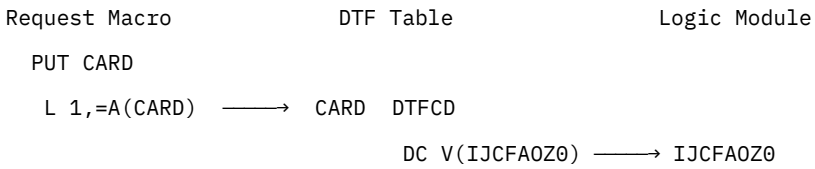


Figure 27. Relationship Between IOCS Macros and Logic Modules

DTFxx Macro

A DTFxx macro must be coded for each logical file that your program wants to refer to by means of a request macro such as GET or PUT, READ or WRITE, or CNTRL. The macro:

- Determines the access method that is to be used by the system.
- Describes the characteristics of the file.

- Indicates the type of processing for the file.
- Specifies the virtual storage areas and routines to be used in processing the file.

If a GET macro is issued, for example, the related DTFxx macro supplies information such as:

- Record type and length.
- Logical unit name of the device from which the record is to be read.
- Address of the area in storage where the record is to be made available for processing by your program.

Figure 28 on page 35 shows an example of a DTFMT macro; it defines a file on magnetic tape.

```
                                                  Column 72
OLDMSTR DTFMT   BLKSIZE=400,                         X
                DEVADDR=SYS001,                      X
                EOFADDR=EOFMSTR,                     X
                FILABL=STD,                          X
                IOAREA1=AREAONE,                     X
                ERROPT=CKOLDBLK,                     X
                IOAREA2=AREATWO,                     X
                IOREG=(3),                           X
                LABADDR=CKOLDBLK,                    X
                RECFORM=FIXBLK,                      X
                RECSIZE=80,                          X
                REWIND=UNLOAD,                       X
                TYPEFLE=INPUT,                       X
                WLRERR=REG6
```

Figure 28. Sample DTFMT Macro

As the example in Figure 28 on page 35 shows, a name is specified in the name field of the definition macro. A DTFxx macro must always have a name in its name field; the I/O request macros for the defined file (such as OPEN, CLOSE, GET, and PUT) must specify this name as an operand.

The Access Methods of IOCS

The DTFxx macro you use in your program depends on the type of device to be used and on how your data is to be accessed:

- Processing with the sequential access method (SAM):

  SAM processing applies when records are to be processed one after the other. Processing files on the various types of devices by using SAM is described in topics as follows:

  – Chapter 3, “Defining and Processing a File with SAM,” on page 41
  – Chapter 4, “Processing a Disk File with SAM,” on page 57
  – Chapter 5, “Processing a Tape File with SAM,” on page 65
  – Chapter 6, “Processing a Unit Record File with SAM,” on page 79
  – Chapter 7, “ Processing a Device-Independent System File with SAM,” on page 95
  – Chapter 7, “ Processing a Device-Independent System File with SAM,” on page 95

  Chapters 4 through 8 expand on Chapter 3.
- Processing with the direct access method (DAM):

  The use of DAM is indicated if the record locations of a file on disk are to be accessed directly. In your program, you must define the file by coding a DTFDA macro. Processing with DAM is described in Appendix B, “Direct Access Method (DAM),” on page 189.
- Processing with the indexed sequential access method (ISAM):

  ISAM does not support the full range of VSE-supported disk devices. Therefore, consider using the licensed IBM program VSE/VSAM instead of ISAM.

  Programs written for processing ISAM files can be used in a VSAM environment through the ISAM Interface Program. For reference purposes, the DTFIS macro is still described in z/VSE System Macros Reference.

The access methods referred to above comprise the logical IOCS.

The DTF Table

A DTFxx macro generates a DTF table. This table contains descriptive information about the related file.

Generally, there is no need for an application program to access fields within a DTF table. However, should you need to reference a field within a DTF table in your program (for example to test error information in the CCB or a certain field of the DTF table), you can reference this table by using the symbol

```
filenamex
```

where x is a letter used to access a field of interest.

When you reference the DTF table, ensure addressability through the use of a base register.

Logical Units

The technique of using symbolic rather than actual device-addresses allows you to write your program as if all devices were available. Instead of defining the device address in the program, you supply a logical unit name as a symbolic address. When the program is running, the logical unit can be assigned to a free device of the required type. In a DTFxx macro, you specify the name of a logical unit in the DEVADDR operand.

A logical unit name has the form SYS*xxx*, and you must choose this name from a fixed set of names as follows:

SYSRDR
Applies to: card reader, magnetic tape unit, or disk extent; used primarily as input unit for job control statements.

SYSIPT
Applies to: card reader, magnetic tape unit, or disk extent; used as the primary input unit for programs.

SYSPCH
Applies to: card punch, magnetic tape unit, or disk extent; used as the primary unit for punched output.

SYSLST
Applies to: printer, magnetic tape unit, or disk extent; used as the primary unit for printed output.

SYSLOG
Applies to the operator console used for communication between the operator and the system. Can also be assigned to a printer.

SYSnnn
Represents all the other (programmer) logical units that can be used under VSE. These units range from SYS000 to SYS254. Figure 29 on page 37 shows how the logical unit you supply in your program relates the program to an I/O device of your system.

Each of these programmer logical units can be assigned to any partition without a prescribed sequence, except when using DAM, where EXTENT job control statements must be supplied and the logical unit names for multivolume files must be assigned in consecutive order.

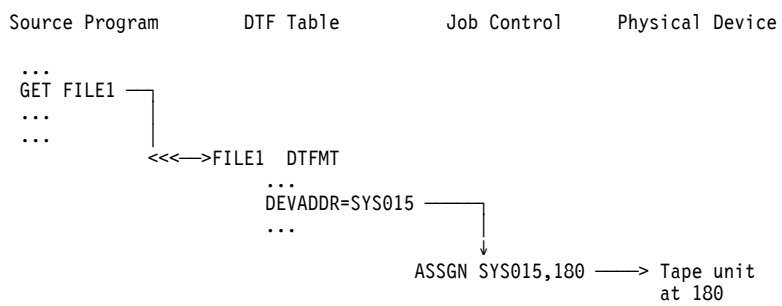


Figure 29. Relationship Between Request Macro, DTF Table, and I/O Device Assignment

For extents on different volumes, different logical units must be assigned for a file. For files processed by SAM or DAM, only one logical unit can be assigned to all extents of a file on one volume.

For a file on disk, you can supply the logical unit name in the job control EXTENT statement. The system uses this logical unit name, if you supply such a name also in the DEVADDR of the DTFxx macro.

A logical unit is assigned to a device by way of the job control ASSGN statement or command. A program that processes tape records, for example, can call the tape device SYS015. When the program is about to be executed, the operator can assign (by an ASSGN command) any available tape drive to SYS015.

It is possible to assign several logical units to the same device. Disk devices, for example, can be used at the same time by several programs. For more details about sharing devices, refer to z/VSE Guide to System Functions.

Logic Module Generation (xxMOD) Macro

Each DTF table, except DTFCN and DTFPH, must link to an IOCS logic module. More than one DTF table can link to the same logic module. Figure 27 on page 34 shows how linkage is established between a request macro for a file and the associated logic module.

A logic module contains the code needed to perform the I/O functions required by your program. For example, it reads or writes data, tests for unusual input/output conditions, blocks or deblocks records if necessary. Most IOCS request macros use a logic module to perform the requested function.

Providing Logic Modules

You need not code logic modules for:

- DTFSD and DTFDA files.
- DTFDI files on disk or for output to an IBM 3800 printer or a printer of the class PRT1.
- DTFMT files.
- DTFPR files for output to an IBM 3800 printer or a printer of type PRT1.

Your VSE system includes preassembled logic modules for those files. These modules are automatically loaded during IPL and linked to your program during OPEN processing for the file.

For any other file, IOCS requires you to code the logic module generation macro (xxMOD) that corresponds the DTFxx macro for the file. You can assemble the macro in-line with your program or separately. If you assemble the macro separately, you must submit the assembled module as input when you link-edit your program. This is further discussed in Appendix A, “Assemble and Link-Edit Programs Using IOCS,” on page 181.

If the required standard logic modules are cataloged in a sublibrary accessible by the linkage editor, you need not code the generation macros in your program. Instead, you can autolink the required modules from this sublibrary when you link-edit your program.

Some of the module functions are included selectively in accordance with the operands you specified in the xxMOD macro. If you code the macro yourself, you can select or omit certain functions such that

the generated module meets the requirements of your program. If you do not code the xxMOD macro yourself, IOCS automatically selects or omits the appropriate functions.

Note: If you issue a request macro (such as WRITE or CNTRL) for a file, and the associated module does not include the desired function, then the system cancels your job with an illegal-supervisor-call message.

Subsetting/Supersetting of Logic Modules

The type of logic module to be used is determined by the functions that are to be supported. Theoretically, this means that a specific logic module may have to be generated for each DTFxx macro. Most likely, a number of these modules would be a subset module to a superset module. A superset module is one that performs all the functions of its subset (or component) modules. Using a superset module thus avoids duplication and saves storage space.

The functions required for several similar files (several DTFPRs, for example) are thus provided by a single xxMOD macro, although the DTFxx macros have different operands. An example of the functions which are included in a superset module replacing two subset modules is shown below for a PRMOD macro:

Functions of Superset Module	Functions of Subset Module 1	Functions of Subset Module 2
• Optional use of CNTRL macro. • Work area and I/O area processing. • Support of printer overflow. • Support of specified error actions.	• CNTRL macro cannot be used. • Work area and I/O area processing. • No printer overflow support. • Support of specified error actions.	• Optional use of CNTRL macro. • I/O area processing only. • Support of printer overflow. • Support of specified error actions.

Supersetting and subsetting as described above does not apply to logic modules that are shipped with the system. Any logic-module generation requests for such modules in your program are ignored.

The subsequent paragraphs discuss the superset/subset technique in more detail.

Note: No subsetting/supersetting takes place if you code an xxMOD macro for each DTF of a given device type.

If you do not code the logic modules yourself, IOCS (together with the linkage editor) automatically performs all subsetting and supersetting that is possible.

If you code the logic modules yourself, subsetting/supersetting can be achieved by coding a single xxMOD macro containing all of the functions needed by all of the DTFs which use that macro. Proceed in either of the following ways:

Do Not Name the Module

This means that you specify neither a name for the xxMOD macro nor a MODNAME operand in your DTFxx macro. IOCS will name the module for you.

IOCS generates a standard module name based on the functions required by the related DTFxx macro. Likewise, if you code your own logic module and omit the name from the name field, IOCS generates a standard module name according to the operands defined in the xxMOD macro.

IOCS performs subsetting/supersetting of modules with standard module names for a program's DTFxx macros for the same type of file. IOCS simply includes the required services in one single module. How IOCS forms subset/superset names is shown by charts under "Subset/Superset xxMOD Names" following the discussion in z/VSE System Macros Reference. The figure gives an example of these charts.

```
          a  b  c  d  e
     ---------------------
          *  +  +  +  *
     I  J  x  F  B  C  W  Y
          U  Z  Y  Z  Z
          V  +  +
          W  E  N
             Z  S
                Z

+  Subsetting/supersetting permitted.

*  No subsetting/supersetting permitted.
x  DTF-dependent character.
```

Figure 30. Example of a Subset/Superset Naming Chart

The letters represent functions which can be performed by the logic module. z/VSE System Macros Reference relates these functions to the corresponding letters.

If a module name is composed of letters from the top row exclusively, it can be only a superset name. A module containing a W in column d, for example, is a superset of a module with a name containing a Z in this column. This superset module therefore performs also the functions of the Z type module.

Other examples of subsetting and supersetting: module IJxWESZZ is a subset module to the superset module IJxWENZZ; IJxWEZZZ is another subset module to superset module IJxWENZZ; it is also a subset module to superset module IJxWESZZ.

An asterisk (*) indicates that for the letters beneath no subsetting or supersetting is permitted, while a plus (+) sign in a column indicates that subsetting/supersetting is permitted. Two plus signs in a single column divide that column into mutually exclusive sets. In the example shown in Figure 30 on page 39, C is only the superset of Y. C is not a superset of N, S, or Z. Conversely, N, S, or Z is not a subset of C.

The vertical arrangement of letters within a column is always in alphabetical order. If a column is divided by plus or asterisk signs into sets, then the vertical arrangement of letters within each set of a column is in alphabetical order.

Name the Module

This means that you specify a name for the xxMOD macro. In addition, you must specify this name also in the MODNAME operands of all the DTF macros that refer to that module.

Subsetting/supersetting occurs if one module contains all of the functions needed by all of the DTF tables that are to use the module.

Nothing is gained by giving your modules standard IOCS names. Should you decide to name your modules yourself, use names that are meaningful in the context of your program.

Logical IOCS Versus Physical IOCS

As mentioned under “The Access Methods of IOCS” on page 35, the following access methods comprise the logical IOCS (LIOCS):

- Sequential access method (SAM)
- Direct access method (DAM)
- The indexed sequential access method (ISAM) also belongs to LIOCS but should no longer be used. Use VSE/VSAM instead.

As opposed to LIOCS, physical IOCS (PIOCS) allows you to control the transfer of data by channel programs that you have coded into your program.

To control data transfer on a PIOCS level, VSE requires that you define your file using the DTFPH macro if either standard labels are to be checked or written or if the file resides on a direct access device.

Processing with PIOCS is described in Appendix C, “Processing a File with Physical IOCS (PIOCS),” on page 211.

Chapter 3. Defining and Processing a File with SAM

You use the sequential access method (SAM) if you want to retrieve the records of a file as they are stored, one record after the other. You use SAM to write records into a file as they are built by your program, again one record after the other.

To process a file, SAM requires that you

1. Define the file's characteristics using the proper DTFxx macro.
2. Create the required IOCS logic module, if necessary.
3. Open (activate) the file. You use the OPEN macro for this purpose.
4. Issue suitable I/O request macros (GET or PUT, for example) at those points in your program where the requested function is to be performed.
5. Close (deactivate) the file after all records of the file have been processed. You use the CLOSE macro for this purpose.

Defining the Characteristics of a File

You define a SAM file with the appropriate DTFxx macro. In this macro, you describe the characteristics of your file. These characteristics are, for example:

- The length and format of its records
- The device type on which it is stored
- The operations that you want to be performed besides reading or writing records. These are operations such as writing a tape mark or reading or writing a short block, space a line before printing.
- The handling of errors detected by SAM.

Table 4 on page 41 lists the DTFxx macros used for SAM processing. It includes the related xxMOD macros you can use to define an IOCS logic module. The macros are listed by supported device classes in alphabetical order.

Table 4. SAM DTFxx and xxMOD Macros by Supported Device Classes

File to be processed	Declarative Macro	I/O Module
Card I/O device	DTFCD	CDMOD
Console display	DTFCN	-
Device independent	DTFDI	DIMOD (see Note 1)
Disk, sequential I/O	DTFSD	-
Magnetic tape	DTFMT	-
Printer	DTFPR	PRMOD (see Note 2)

Note:

1. Not needed for a disk device, a PRT1 printer, or an IBM 3800.
2. Not needed for a PRT1 printer, an IBM 3800, an IBM 4248 in native mode, or a 6262 printer.

The following sections describe the most commonly used DTFxx macro operands. However, not all of these operands apply to all file types. For more details about the operands you can specify, see the topic on the device type you are using, or consult z/VSE System Macros Reference.

File Type (TYPEFLE)

You define the processed file type with the TYPEFLE specification. Specify one of the following:

TYPEFLE=INPUT
The file is used for input via the GET macro.

If the file resides on disk, you can update this file by specifying also UPDATE=YES. In this case, records are read, processed, and then written back into the same record location from which they were read. For more details, see "Processing an Update File" on page 60.

TYPEFLE=OUTPUT
The file is used for output via the PUT macro.

TYPEFLE=WORK
Specifies a tape or disk file that can be used as a work file. A work file can be used for both input and output. Thus it is suitable for passing on intermediate results from one phase to another or from one job step to another. The file may be used by just one phase to read data from or write data into it. For more details, see Chapter 3, "Defining and Processing a File with SAM," on page 41.

Record Format (RECFORM)

SAM needs to know the format of your records. You define this format with the RECFORM=format operand. For format, you can specify one of the following:

FIXUNB
For fixed-length unblocked records.

FIXBLK
For fixed-length blocked records.

VARUNB
For variable-length unblocked records.

VARBLK
For variable-length blocked records.

SPNUNB
For spanned variable-length unblocked records.

SPNBLK
For spanned variable-length blocked records.

UNDEF
For undefined records.

For fixed-length blocked, spanned or undefined output records, you must also specify the RECSIZE operand.

The record formats that you can use are discussed under "Record, Block, and Control Interval" on page 12.

Record Size (RECSIZE)

How you define the size of the records of a file depends on the record format. For example, if the record format is defined with:

RECFORM=FIXBLK
Then specify the number of bytes (characters) of each record in RECSIZE=n.

RECFORM=UNDEF or RECFORM=SPNxxx
Then specify RECSIZE=(r), where r is a register that contains the length of each record.

Before you issue a PUT macro for an output file of undefined or spanned records, load the length of the record into the register specified by r. For input files, SAM places the length of the record transferred to virtual storage into the specified register.

RECSIZE is invalid for work files. You specify the length of the record in the READ or WRITE macro.

I/O Area Definition (IOAREA)

When reading or writing a file, SAM transfers data between an I/O device and an area in virtual storage. You specify the name of this area with the operand IOAREA1=name. For name, specify the symbolic address that you used in your program to reserve this area of storage.

You use the BLKSIZE operand to define the length of this area. This operand is discussed separately in the next section.

Following is an example of an I/O-area definition.

```
FILEA DTFSD IOAREA1=BUF,BLKSIZE=200
      ...
BUF   DC   CL200' '
```

To improve device performance, you can use a second I/O area, which you define with the operand IOAREA2=name. If you do this, SAM transfers records alternately to or from each area.

If you do not define an I/O area for a SAM file in the DTFSD, the system sets up this area in the partition GETVIS space. Note, however, that IOAREA1 is required for other types of files (for example, DTFMT).

I/O Area Length (Block Size BLKSIZE)

You define the I/O area length with the BLKSIZE operand. The length you specify depends on a number of factors such as the record format, the blocking factor, and the device on which the file is stored.

Specify the length of your largest block of records, including the block and record descriptors, if the record format is variable. Specify the length of your largest record if the record format is undefined. For the smallest and largest values that you can specify for BLKSIZE, refer to the description of the applicable DTFxx macro in z/VSE System Macros Reference.

For optimum use of the storage capacity of a disk, you can specify BLKSIZE=MAX. This causes SAM to set the I/O area to a length as follows:

- If your file resides on a CKD disk, to a full track of the device on which the file resides.
- If your file resides on an FBA device, the operand specifies the logical block size. For an FBA disk, the maximum value is 32 761 (the maximum CISIZE value minus 7). In that case, SAM sets the length of the I/O area to the highest even multiple of RECSIZE that is smaller than 32 761 bytes.

I/O Register Specification (IOREG)

You must specify this operand if one of the following is true:

- Two I/O areas without a work area are used.
- No I/O area has been specified (not for DTFMT files).
- Blocked records are processed in the I/O area.
- Undefined or variable-length magnetic tape records are read backwards.

SAM puts into this register the address of the logical record that is available for processing. This is the address of the next available input record or of the area where you can build the next output record.

If you omit the IOAREA operand (to let OPEN obtain an I/O area of the required length), OPEN returns the address of the acquired I/O area in the IOREG register. Note, however, that IOAREA1 is required for other types of files, for example, DTFMT.

For an output file with variable-length blocked records, an additional register must be specified in the VARBLD operand if the records are built in the I/O area(s). This register contains the length of the available space remaining in the output area each time SAM has processed a PUT macro for the file.

If you use a work area instead of an I/O area, specify WORKA=YES and do not specify IOREG.

Work Area Specification (WORKA)

You may wish to process your records in a work area separate from the I/O area(s), especially if you are using blocked records. You accomplish this by:

1. Including the operand WORKA=YES.
2. Setting up the work area in virtual storage (with a DS instruction, for example). For variable-length records, the work area must be large enough to hold the longest logical record.
3. Specifying the (symbolic) address of the work area in the I/O request macro.

SAM moves the record from the specified I/O area to the specified work area, and vice versa.

The WORKA operand is required for spanned records; the work area must be long enough to hold the longest spanned record.

When you use a work area, do not specify the IOREG operand.

The example in Figure 31 on page 44 shows how to define I/O- and work areas. The GET macro causes SAM to read blocks of 500 bytes alternately into areas A1 and A2. Once a block has been read into an I/O area, SAM moves a logical record of 100 bytes into the work area A3.

```
                                                Column 72
FNAME      DTFMT                                 X
                  IOAREA1=A1,                    X
                  IOAREA2=A2,                    X
                  WORKA=YES,                     X
                  BLKSIZE=500,                   x
                  RECSIZE=100,                   X


A1         DS     500C
A2         DS     500C

           GET    FNAME,A3 ──┐
┌─────────────────────────────┘
│
↓
A3         DS          100C
```

Figure 31. Coding Example for the Use of a Work Area

Error Handling (ERROPT, WLRERR, and ERREXT)

The operands that you may use for processing I/O and record-length errors are ERROPT, WLRERR, and ERREXT. Not all of these operands apply to all SAM files. For complete information, refer to the subsequent topics or to z/VSE System Macros Reference

The ERROPT Operand

This operand specifies how SAM is to handle read or write errors. The operand has three specifications:

ERROPT=IGNORE
For an input file, the error condition is ignored and the error record is made available for processing. When reading from a file of spanned records, SAM passes to your program the complete record (or a block of spanned records) rather than just the one physical record in which the error occurred.

For an output file, the error is ignored and the physical record containing the error is treated as a valid record. If the file consists of spanned records, SAM writes any remaining record segments, if this is possible.

ERROPT=SKIP
For an input file, no records in the error block are made available for processing. The next block is read and processing continues with the first record of that block.

When reading from a file of spanned records, SAM skips the complete spanned record (or block of spanned records) and not only the physical record in which the error occurred.

On output, the error is ignored and the physical record containing the error is written as a valid record. If the file consists of spanned records, SAM writes any remaining spanned record segments, if this is possible.

ERROPT=name
SAM gives control to the routine whose (symbolic) address you specified for name. In this routine, you can process (or make note of) the error condition as desired. For more information about user-written error routines refer to the sections discussing error handling in the topics that deal with the processing of the various types of files.

The WLRERR Operand

This operand specifies the name of your routine that is to receive control when a wrong-length record is read. The operand applies to input files only.

If you omit this operand, and SAM detects a wrong-length record, one of the following occurs:

- If the ERROPT operand is also omitted, the wrong-length error condition is ignored.
- If the ERROPT operand is included for this file, the wrong-length record is treated as an error block and is handled according to your specification for an error. The operand is discussed under "The ERROPT Operand" on page 44.

The ERET Macro

You can use the macro to return control from your ERROPT or WLRERR exit routine. In this macro, specify the action to be taken: SKIP, IGNORE, or RETRY.

For a tape file, you must include ERREXT=YES in the DTFxx macro to return control to SAM. As an alternative, you can return control by a branch to the address contained in register 14.

End-of-File Exit (EOFADDR)

This operand specifies the name of your end-of-file routine. SAM automatically branches to this routine on an end-of-file condition. In this routine, you can perform any operations required at end of file. Generally, you issue a CLOSE macro.

Opening a File for Processing

To open a file (or make a file ready) for processing, you use the initialization macro OPEN. As an operand of the macro, specify the name of the file that is to be opened or the register that points to this name in storage. Figure 32 on page 46 shows a skeleton source program which includes an OPEN macro for a tape file named FNAME.

```
                                                        Column 72
FNAME      DTFMT    ...                                     X
                    IOAREA1=A1,                             X
                    IOAREA2=A2,                             X
                    WORKA=YES,                              X
                    BLKSIZE=500,                            x
                    RECSIZE=100,                            X
                     ...
                     ...
A1         DS       500C
A2         DS       500C
           ...
           OPEN     FNAME
           ...
GETNXT     GET      FNAME,A3
           ...
           B        GETNXT
A3         DS       100C
           ...
```

The OPEN for a file must precede the first request macro for the file.

Figure 32. Coding Example for Opening and Accessing a File

Before you open a file, ensure that it is not already open.

In your program, you can open up to 16 files with one OPEN macro. These files can use any combination of access methods. You specify the name of the file to be opened either symbolically or by using register notation.

The OPEN macro, together with the ASSGN job control statement, associates the logical file defined in your program with the actual data file on the assigned I/O device. The link established between your logical file and the data file is retained until your program issues a completion macro, which is discussed under “Opening a File for Processing” on page 45.

If your program closes a file but needs it again for additional processing, then the program can issue another OPEN for the file.

If OPEN attempts to open a file whose device is unassigned the job is canceled.

OPEN checks or writes standard IBM disk and tape labels. However, if you open a file to process user standard labels (UHL or UTL) on disk or on tape, or non-standard labels on tape, your program must provide the information for checking or building the labels. If this information is obtained from another input file, open that input file before you open the labeled file. Either specify the name of this input file ahead of the name of the labeled file in the same OPEN or issue a separate OPEN macro for the input file.

To ensure that SAM properly processes labels on disk or tape, you must supply job control label-information statements:

- For a file on disk:
  - One DLBL statement
  - One or more EXTENT statements
- For a file on tape – A TLBL statement.

Processing of labels is discussed separately for the processing of files on the various devices:

- For a file on disk in the section “Processing of Labels” on page 58.
- For a file on tape in the section “Processing of Labels” on page 65.

Refer to z/VSE System Macros Reference how to code a label processing routine.

Self-relocating programs must use OPENR to open a file. OPEN and OPENR perform essentially the same functions. When OPENR is used, the generated addresses are self-relocating. Throughout the publication, the term OPEN refers also to OPENR, unless stated otherwise.

Reading (GET) and Writing (PUT) of Data

SAM allows you to store and retrieve data without coding your own blocking and deblocking routines. In your program, you can use one or two I/O areas; you can process the records in a work area or in the I/O area(s).

When a file on an FBA disk is to be processed, SAM uses a single control interval (CI) buffer to transfer data between virtual storage and the device. If a work area is specified, read requests for a logical block cause data to be moved from the CI buffer directly to the work area; an I/O area, if specified, is ignored. When no work area is used, data is transferred from the CI buffer to the I/O area(s).

Obtaining a Record for Processing (GET)

The GET macro makes the first (or next) logical record from an input file available for processing.

In your program, you can use GET for any input file defined by a SAM DTFxx macro, and for any type of record. If GET is used for a file that includes checkpoint records, SAM bypasses these records automatically.

A GET macro issued after the last record of an input file has been accessed results in an end-of-file condition.

When a file occupies more than one area on a disk volume, GET automatically switches from one extent to the next. GET checks for an end-of-volume condition and starts automatic volume switching if an input file spans two or more volumes.

If WORKA=YES is specified in the DTFxx macro for your file, all GET macros for this file must specify a work area name or a register that points to a work area. This causes GET to move each individual record to that work area.

You can use several work areas for one file. You can have all records of your file processed in the same work area, or different records of the file in different work areas. In the first case, each GET for the file specifies the same work area. In the second case, different GET macros specify different work areas. It may be of advantage to use two work areas with alternate areas specified in alternate GET macros. To use a work area, specify WORKA=YES in your DTFxx macro, but do not specify the IOREG operand.

The function of SAM for a GET macro differs to a certain extent for the various types of records as discussed below.

- For unblocked records

  If only one input area is used, each GET transfers a single record from the associated I/O device to the input area. The record is then transferred to the work area, if one is used.

- For blocked records

  The first GET macro transfers a block of records from the associated device into virtual storage. In addition, this GET either sets the IOREG register (if one is specified) to point to the first logical record or moves that record to a work area.

  Subsequent GET requests either step up the register, or move the next sequential record to the specified work area, until all records of the block are processed. Another GET makes a new block of records available to your program and either initializes the register or moves the first record.

- For spanned records Your DTFxx macro must include operands as follows:
  - RECFORM=SPNUNB or RECFORM=SPNBLK
  - RECSIZE=(r)
  - WORKA=YES.

  GET assembles spanned record segments into logical records in your work area. The length of the logical record is passed to you in the register specified in the RECSIZE operand. A logical record may span disk extents, but it cannot span disk volumes.

- For undefined records Your DTFxx macro must include the operands:

– RECFORM=UNDEF
– RECSIZE=(r)

GET treats the records as unblocked. Your program must locate (deblock) individual logical records if two or more logical records make up one undefined record.

In the specified RECSIZE register, SAM stores the length of the record it has read. Other than that, no characteristics of the record are known or assumed by SAM. Your program must determine these characteristics.

Storing a Record after Processing (PUT)

The PUT macro writes a logical record to the associated output device. In your program, you can use the PUT macro for any output file defined by a SAM DTFxx macro, and for any record type. PUT operates very much like a GET, but in reverse. You issue a PUT after a record has been built.

PUT checks for an end-of-volume condition and starts automatic volume switching if an output file spans two or more volumes.

If you build the records of a file in a work area, PUT moves each record from the work area to the output (I/O) area or, for output to an FBA disk, to the CI buffer. All PUT macros for the file must specify a work area.

When a PUT macro processes a logical record, this record remains in the output (or work) area until it is cleared or replaced by other data. SAM does not clear the area. Therefore, if you plan to build another record whose data does not use every position of the area, ensure that you clear this area before you build the record. If you use a work area in your program, make it a point to build your records only in that area and never change the contents of the I/O area.

The function of SAM for a PUT macro differs to a certain extent for the various types of records as discussed below.

- For unblocked records

  Each PUT transfers a single record from the output area to the output device. If a work area is specified in the PUT macro, the record is first moved from the work area to the output area and then to the device.

- For blocked records

  The individually built records must be formed into a block in the output area before the block can be transferred to the output device.

  Fixed-length blocked records can be built directly in an output area or in a work area. Each PUT macro for these records either steps up the IOREG register or moves the completed record from the specified work area to the next free record location in the output area or CI buffer. When an output block of records is complete, the next PUT macro causes the block (or CI) to be transferred to the output device. This PUT resets the IOREG register, if one is used.

- For variable-length records

  You can build records of this type in an output area or in a work area. Your program must (1) determine the length of the record that is being built and (2) insert this length in the first two bytes of the four-byte record descriptor.

  If your records are to be blocked and you build them *in a work area*, PUT checks the length of each output record to ensure that the record fits into the remaining portion of the output area or CI buffer. If the record fits, PUT immediately moves the record. If the record does not fit, PUT causes the completed block to be written and then moves the record from the work area to the output area.

  If you build the records *in the output area*, the DTFxx VARBLD operand, the TRUNC macro, and additional programming are required. Before it starts processing for the next record, your program must find out whether that record will fit into the remaining portion of the output area. The amount of space available in the output area is supplied to your program in the register specified in the VARBLD operand. Each time a PUT request is complete, SAM passes in this register the number of bytes remaining in the output area. Your program can compare the length of your next record with the number of free bytes.

If the record does not fit, you program must issue a TRUNC macro to transfer the completed block of records to the output device. The entire output area is available for building the next block after SAM has finished processing the TRUNC macro.

- For spanned records

  PUT divides the records built in the work area into record segments according to the length specified in the BLKSIZE operand. For disk output, spanned records must not span volumes. If there is not enough space on the current volume to contain a spanned record, SAM attempts to write the entire spanned record on the next volume.

- For undefined records PUT treats records of this type as unblocked. Your program must provide for blocking of logical records, if this is desired. It must also determine the length of each record and load this length into a register before issuing the PUT macro for that record. The register used for this purpose must be specified by the RECSIZE operand of the DTFxx macro for the file.

For an *update* file (applies to data on disk), SAM repositions the device to the first logical record of the block. This allows a record of the file to be read, modified, and written back to the same location from which it was read. For more details, see "Processing an Update File" on page 60.

Processing Blocked Records Selectively

Frequently, a program processes a file, record after record, starting with the first logical record. Consider blocked records in this case, especially since:

- Blocking or deblocking is done automatically by the GET and PUT macros.
- Blocking of records results in a more effective use of the involved I/O device.

The use of blocked records offers possibilities that do not exist when records are unblocked. When processing a logical record of a block, your program can, for example, request SAM to ignore the remaining records of that block and obtain the first logical record of the next block. For output, your program can request SAM to skip the remainder of the current block and use the next logical record as the first one of the next block. For a file with blocked spanned records, your program can bypass all subsequent records of the block being processed and obtain the first segment of the next logical record in the new block.

An application could benefit from these possibilities, for example, if it must process a file that consists of several major groups of logical records. If each category starts with a new block, it is easy to locate any of the categories for selective processing. Only the first record of each block would have to be checked.

To achieve this, you would use the RELSE (release) macro with input, and the TRUNC (truncate) macro with output:

- The RELSE (release a block) macro

  The macro is used with blocked records on disk or on tape. It causes the next GET to ignore any logical records remaining in the current block and to get the first logical record of the next block. This applies also to blocks of spanned records.

- The TRUNC macro

  The macro is used with blocked records to be written to disk or to tape. It allows your program to write a short (truncated) block of records. A truncated block does not include padding.

  TRUNC causes the next PUT macro to regard the output area as full and, subsequently, the next logical record to be placed into the next block. Thus, the TRUNC macro can be used for output in a way similar to the RELSE macro for input. When the end of a category of records is reached, the current block can be written, and the new category can be started at the beginning of a new block.

  If the TRUNC macro is issued for fixed-length blocked records of a file on disk, your DTFSD macro must include TRUNCS=YES. If your file resides on an FBA disk, TRUNC causes an immediate write to the FBA disk only if also PWRITE=YES is specified.

  The last record written in a short block is the record that was built and included in the output block by the last PUT preceding the TRUNC macro. Therefore, if your program builds the output records in a work

area and finds that a record belongs into a new block, it must first issue a TRUNC to have the current block written to the I/O device. The subsequent PUT for the record in the work area then moves this record into the new block. If your program builds the output records in the output area, the program must determine whether a record belongs in the block before it builds the record.

The TRUNC macro must be used to write a complete block of records if variable-length blocked records are built in the output area. Your program must check the VARBLD register to determine whether the next variable-length record fits into the current block. If the record does not fit, your program must issue the TRUNC macro to have the current block transferred. This makes the entire output area available to build the next record.

Processing a Work File

Normally, a work file is used to pass intermediate results between successive phases or job steps. A work file can be written, read, and rewritten by just one phase, and the file need not be closed and reopened.

The READ and WRITE work-file macros provide for overlapped processing. While a record is being transferred, your program can perform other operations that do not depend on the presence of the record. To ensure that the READ or WRITE has been completed, you must code the CHECK macro in your program preceding the instructions that use the I/O (or work) area.

The NOTE, POINTR, POINTS, and POINTW work-file macros allow you to do a certain amount of direct processing. You can position the file to a certain point and continue sequential processing from that point.

There are certain restrictions for the use of a work file:

- It can be specified only as having unblocked records of fixed lengths or of undefined format. To use blocked records with a work file, do either or both of the following:
  - Code your own blocking and deblocking routines.
  - Place the file on an FBA device, if one is available, to have SAM use a CI buffer.
- It must be contained on a single volume, but not on a tape written in ASCII mode.
- On an IBM 9346 tape device you cannot process a work file that requires previously stored data to be overwritten. A unit check with command reject indicated in the sense byte occurs.
- It cannot be extended.
- Automatic I/O area switching is not possible; your program must supply the address of your I/O area each time it issues a READ or WRITE macro.
- Only data-area extents (type 1 and also type 8, split extents) are supported.
- If you use the CI format, the number of logical blocks per control interval is limited to 255.

Figure 33 on page 50 shows an example for defining a work file on disk.

```
                                                    Column 72
FILEW    DTFSD      BLKSIZE=150,                        X
                    TYPEFLE=WORK,                       X
                     ...
                    EOFADDR=ENDFRTN,                    X
                    RECFORM=UNDEF,                      X
                    UPDATE=YES
```

Figure 33. Example for Defining a Work File on Disk

Opening the File

SAM opens a work file for output.

Work File on Disk

Like any other file, it must have standard file labels. This implies that you must supply the required label information either with the job or stored permanently in the label-information area of your system.

Since a work file is opened for output, reopening the file within a job (for example to pass information to a subsequent job step) causes an overlapping-extent message to be issued to the operator. The operator may then delete the format-1 label, after which the open routines create a new label for the file, and the job continues.

Work File on Tape

If the work file is defined as a standard labeled file (DTFMT FILABL=STD), you must supply the TLBL job control statement. SAM reads or writes the standard header and trailer labels as for any other file with standard labels. However if the file's expiration date has been reached, SAM creates a new label consisting of HDR1 followed by 76 blanks. This marks the file as a work file. However, if the HDR1 file-id and the TLBL file-id are equal, SAM checks the expiration date and, if the file is expired, message EXPIRED FILE appears. With response IGNORE, the old HDR1 label will be updated; with response BYPASS, the old HDR1 label will be kept or, if the file is unexpired, the old HDR1 label will also be kept. If the HDR1 file-id is not equal to the TLBL file-id and if the file is expired, the old HDR1 label will be overwritten or, if the file is unexpired, message UNEXPIRED FILE appears; with response IGNORE, the old HDR1 label will also be overwritten.

Trailer labels are not processed.

If the work file is defined as an unlabeled file (DTFMT FILABL=NO or no FILABL operand), you need not supply any label information.

If the tape does not have standard labels, SAM does not create labels for the file.

If the tape is not positioned at a tape mark when your program issues an OPEN for the work file, SAM writes a tape mark at the current tape position. If this is not desirable, have your program issue a CLOSE for the work file at the end of the job step that uses the file. This ensures that no data is lost when SAM writes the tape mark. In the next job step, your program can use a CNTRL or POINT request to position the tape to the first record to be read. However, the preferred approach is to correctly position the tape before your program of the next job step opens the work file.

Processing the File Sequentially

You use READ and WRITE macros to read a record from or write a record to a work file. Before your program can process a retrieved work-file record, the READ operation for this record must be complete. Before the program can build a new work-file record, the WRITE operation for the preceding work-file output record must be complete. In you program, you use the CHECK macro to halt processing until this transfer of data is finished.

Figure 34 on page 53 shows how you can use the various work file macros that are discussed below.

The READ Macro

The macro causes a record to be transferred from the defined work file to an area in virtual storage.

Specify the length operand of the macro if the records of your work file are undefined. This operand defines the number of bytes to be transferred. Specify 'S' if the entire record is to be read.

If your work file has unblocked records of fixed length, the number of bytes to be transferred is specified in the BLKSIZE operand of your DTFxx macro.

READ can be used also to read backwards from magnetic tape.

The WRITE Macro

The macro requests that a block of data be transferred from an area in virtual storage to an output file. The macro operates in the same way as READ, but in reverse order.

For fixed-length unblocked records, the number of characters to be written is specified in the BLKSIZE operand of your DTFxx macro.

Specify the length operand of the macro if the records of your work file are undefined. This operand defines the number of bytes to be transferred.

The CHECK Macro

The macro avoids that data requested by a READ macro is processed before the transfer of this data is complete. The macro avoids that data being written to a device by a WRITE is overwritten by your program while it builds a new output record. In addition, the macro tests for errors and exceptional conditions that may have occurred during the data transfer.

Use the CHECK macro after each READ or WRITE before you issue any other macro for the same file, or before the contents of the input or output area in virtual storage are changed.

If the data transfer is completed without any error condition, SAM returns control to the next instruction of your program. If the operation results in a read error, SAM processes the option specified in the ERROPT operand. If an end-of-file condition occurs, SAM passes control to the routine specified in the EOFADDR operand.

Note: An end-of-file record is written only if the last operation before the CLOSE macro was a WRITE SQ; a CLOSE macro following a WRITE UPDATE protects the updated file by not writing an end-of-file record.

Processing the Records of the File Selectively

Both READ and WRITE operate strictly sequential, starting either at the beginning of a work file or at a given point to which the file can be positioned. To position a work file to a certain block, use the NOTE and the POINTx macros.

Figure 34 on page 53 shows how you can use the NOTE and POINTS (one of the POINTx macros) in an application program. The subsequent paragraphs discuss the macros in more detail.

```
                                                        Column 72
      FILEW    DTFSD    BLKSIZE=150,                        X
                        TYPEFLE=WORK,                       X
                         ...
                        EOFADDR=ENDFRTN,                    X
                        RECFORM=UNDEF,                      X
                        UPDATE=YES
(1)                     12,LENGTH
(2)   WRREQ    WRITE    FILEW,SQ,OUT,(12)
                ...        | Processing is not related
                ...        | to the data stored in the
                ...        | area OUT.
(3)             CHECK    FILEW
                ...
                BNZ      WRREQ
(4)             POINTS   FILEW
(5)   RDREQ    READ     FILEW,SQ,IN,S
                ...
(6)             CHECK    FILEW
                ...
                BNZ      RDREQ
                EOJ
```

(1)
Loads the length of the record.

(2)
Causes a record to be written.

(3)
Waits until the record is written into the file WRKFLE.

(4)
Repositions the file to the address of the first record.

(5)
Causes a record to be read.

(6)
Waits until the record has been read into virtual storage.

Figure 34. Example of POINTS Macro with Work File Processing

The NOTE Macro

Your program can issue a NOTE after a READ or a WRITE macro or after the I/O operation was checked for completion by a CHECK macro.

NOTE returns to your program, in register 1, the position of the block just read or written. Store this position in a four-byte field for later use by a POINTR or POINTW macro in your program.

For a work file on a CKD disk, NOTE identifies a record by the record number relative to the beginning of the track. It identifies a record on an FBA disk by the record number relative to the beginning of the file. NOTE returns additional information in register 0:

- For a work file on a CKD disk – the unused space remaining on the track following the end of the identified record.
- For a work file on an FBA disk – the length of the longest logical record that fits into the CI following the identified logical record.

Store this remaining space in a two-byte field for later use by a POINTR or POINTW macro if the macro is to be followed by a WRITE SQ or another NOTE while the file is still positioned on the same CI or track that was pointed to.

For a work file on tape, NOTE identifies a record by its record number relative to the load point.

The POINTS Macro

The macro causes a file to be repositioned to its beginning.

For a tape file, POINTS causes the tape to be rewound to the load point and then to be positioned to the first data block. Labels, if any, are bypassed.

For a disk file, POINTS positions the file to the lower limit of the first extent. A POINTS followed by a WRITE SQ, for example, causes the new record to be written and the remainder of the track or control interval to be erased.

Note: Do not code a WRITE UPDATE following a POINTS.

The POINTR Macro

The macro positions the file to a specific record. Your program can then read this record by issuing a READ macro.

A series of READ macros following a POINTR macro read records sequentially, starting with the record specified in the POINTR macro. Your program can get the address to be specified in the POINTR from the result of a previously issued NOTE macro.

Note: For a work file on tape, do not code a WRITE following a POINTR.

For a work file on disk, if a WRITE UPDATE follows the POINTR macro, the record located by the POINTR is overwritten. If a WRITE SQ follows the POINTR macro, the record after the one located by the POINTR is written, and the remainder of the track (or CI) is erased. On an FBA disk, an SEOF is written immediately after the current CI.

The POINTW Macro

The macro positions the file to a record location as follows:

- For a work file on tape – to read or write a record after the one previously identified by NOTE.
- For a work file on disk – to the location of the record that was read or written immediately before the NOTE macro was issued. If your program issues a:

  WRITE UPDATE
  The record identified by the NOTE is overwritten.

  WRITE SQ
  The record after the one identified by the NOTE is written and the remainder of the track or CI is erased. On an FBA disk, an SEOF is written immediately after the current CI.

  If a POINTW (or a POINTR) macro is issued and the work file records are in undefined format, it may happen that a replacement record longer than the original record cannot be written into the space available on the track (or in the CI). In this case, when the next WRITE is performed, the original record remains as the last record on the track, and the replacement record is written as the first record on the next track (in the next CI) of the file.

A series of WRITE macros following a POINTW write records sequentially, starting at the location following the record specified in the POINTW macro. Your program can get the address to be specified from the result of a previously issued NOTE.

A READ macro may follow a POINTW macro. In this case, SAM reads the record identified by the NOTE macro.

Retaining or Deleting a Work File

If you want to retain a work file on disk for later use, do the following:

1. Specify DELETFL=NO in your DTFSD macro.
2. Ensure that the file's expiration date has not yet been reached.

This causes SAM to retain the file until the expiration date is reached.

To have SAM delete a work file after use, simply omit the DELETFL=NO operand. Another job requiring a work file can then use the same extents and the same file name.

Requesting a Non-Data Device Operation

The CNTRL (control) macro allows you to specify commands for serial I/O devices. However, if you issue the macro for a sequential file on disk, SAM ignores the macro.

In the macro you can specify such functions as tape rewinding for a tape drive and line spacing or page ejects for a printer.

The CNTRL macro does not wait for completion of the command before returning control to you. The permitted mnemonic codes are device-dependent. For a list of these codes, refer to z/VSE System Macros Reference.

Closing the File for Processing

The CLOSE macro ends the association of the logical file defined in your program with a file of data on an I/O device. Issue a CLOSE macro to close a file that was opened by an OPEN macro, except for a console (DTFCN) file. Up to 16 files can be closed with one CLOSE.

Once SAM has processed a CLOSE for a file, your program can issue no further requests for the file until this file is opened again.

A CLOSE normally closes an output file by writing an EOF record and output trailer labels, if any.

If your program issues a CLOSE for an unopened tape-input file, SAM performs the option specified in the REWIND operand of the DTFMT. For an unopened tape-output file, a CLOSE does not cause a tapemark or any labels to be written. Also, no REWIND option is performed.

As with an OPENR macro, you must use the CLOSER macro if your program is to be self-relocating. The CLOSE and the CLOSER macros are essentially the same, except that when CLOSER is specified, the addresses generated from the parameter list are self-relocating. Throughout the publication the term CLOSE refers also to CLOSER, except where stated otherwise.

IOCS Request Macros Used with Declarative Macros

Table 5 on page 55 concludes the discussion of defining a file for SAM processing. It shows which imperative macros you can use to process the records of a SAM file.

Table 5. IOCS Request Macros Used with SAM Declarative Macros

Request Macro	D T F ...									
	CD	**CN**	**DI**	**DR**	**DU**	**MR**	**MT**	**OR**	**PR**	**SD**
CHECK						X	X *			X *
CLOSE(R)	X		X	X	X	X	X	X	X	X
CNTRL	X			X			X	X	X	X
DISEN						X				
DSPLY								X		
ERET					X		X			X
FEOV							X +			
FEOVD										X
FREE										X *

Table 5. IOCS Request Macros Used with SAM Declarative Macros (continued)

Request Macro	D T F ...									
	CD	**CN**	**DI**	**DR**	**DU**	**MR**	**MT**	**OR**	**PR**	**SD**
GET	X	X	X		X	X	X +	X	X	X +
LBRET							X			X
LITE						X				
NOTE							X *			X *
OPEN(R)	X		X	X	X	X	X	X	X	X
POINTx							X *			X *
PRTOV									X	
PUT	X	X	X		X		X +		X	X +
PUTR		X								
RDLNE								X		
READ				X		X	X *			X *
RELSE							X			X
RESCN								X		
SEOV							X			
SETDEV				X						
TRUNC							X			X
WAITF				X		X		X		
WRITE				X		X	X *			X *

*****
Work files only

+
Data files only

Chapter 4. Processing a Disk File with SAM

This topic expands on information given in Chapter 3, “Defining and Processing a File with SAM,” on page 41. It discusses how to open and close a disk file with SAM. It summarizes the requirements for coding an exit routine that processes disk-labels or an error condition. It includes a section on the processing of a file on an FBA disk.

Defining the File

For your application, it might be desirable to do certain processing if an end-of extent condition occurs. You can code a routine to do this kind of processing.

The operand EOXPTR=name of the DTFSD macro specifies the name of the pointer to your end-of-extent routine. SAM gives control to this routine if the end of the last (or only) extent is reached while writing to an output or a work file.

Opening the File

To make a file available for processing, your program must issue an OPEN macro. How SAM processes the macro is discussed separately for input and output.

For Input

If you have a multivolume file, SAM processes only one extent at a time. Therefore, only one volume need be mounted at any point in time. When all processing for this volume is complete, SAM issues the message

```
4n55A WRONG PACK, MOUNT nnnnnn
```

Your operator can then mount the next volume.

When opening a file, SAM checks the extents specified in the EXTENT statements against the extents stored in the labels of the file in the VTOC. This ensures that the extents exist.

If you specified LABADDR in the DTFSD macro for the file, SAM makes the user standard-header labels (UHL) available to your program for checking, one label at a time. When all labels are checked, the first extent of the file is ready to be processed.

SAM makes the extents of a file available in the order of the sequence number in the EXTENT statements. For an input file, you normally use the same EXTENT statements that were used to build the file.

Note: If EXTENT statements with specified limits are included in the job stream and if an extent was created by a response to message

```
4450A NO MORE AVAILABLE EXTENTS
```

when the file was built, you must supply an additional EXTENT statement on input to have SAM process that extent. If no EXTENT statements are submitted, this additional extent is processed normally.

After having processed the last extent of the volume, SAM makes the user standard-trailer labels available for checking, one at a time. SAM then opens the next volume.

When a new volume of a data-secured multivolume file is opened, SAM makes the first extent of the new volume available. At the same time, the extent(s) on the previous volume become unavailable.

For Output

If your program creates a multivolume file, SAM processes one extent after the other, one at a time. Therefore, only one volume need be mounted at a time. When all processing of a volume is complete, SAM issues the message

```
4n55A WRONG PACK, MOUNT nnnnnn
```

Your operator can then mount the next volume.

When a file is opened, SAM checks the standard VOL1 label and the specified extents for the following:

1. That none of the file's extents overlaps another file's extent.
2. If the file is being built on CKD disks and user-standard labels are to be written, the first extent is at least two tracks long.
3. All extents of the file are of type 1 (prime data) or, for a CKD disk, of type 8 (split-cylinder).

When a file on an FBA disk is opened, SAM stores the CI size in the VTOC (for use when the file is re-opened as an input file).

After having checked the label information against the labels in the VTOC, SAM builds the standard label(s) for the file and writes these label(s) into the VTOC.

If you wish to write your own user-standard labels (UHL or UTL) for the file, define the address of your label routine in the LABADDR operand. SAM reserves the first track of the first extent (or a sufficient number of FBA blocks) for these labels and gives control to your label routine.

When SAM has built the header label(s), the first extent of the file is ready for use. Additional extents, if any, are made available one at a time and in the order of the sequence numbers in the EXTENT statements. When the file's last extent on the mounted volume is filled, your LABADDR routine gets control to build user-standard trailer labels. Then, the next volume specified in the EXTENT statements is mounted and opened.

When a new volume of a data-secured multivolume file is opened, SAM makes the first extent of the new volume available. At the same time, the extent(s) on the previous volume become unavailable. When the last extent on the last volume of the file is processed and an end-of-extent routine is available, SAM passes control to that routine. Else, SAM issues an operator message, and the operator may then either cancel the job or type in an extent at the console and have the system continue processing the job.

Processing of Labels

SAM requires IBM-standard labels for files on disk. Therefore, corresponding DLBL and EXTENT statements must be available to the system when SAM is to open a disk file. For more information about these statements, refer to z/VSE System Control Statements. z/VSE System Macros Reference gives detailed instructions for coding a label-exit routine.

Processing for OPEN

SAM uses the information supplied with the DLBL and EXTENT statements for the file to check the following:

- For an input file

  That its extent(s) coincide with or are within the existing extent(s) as stored in the VTOC.
- For an output file

  That it does not overwrite another, unexpired file stored on the disk volume to which your file is to be written. If your file would overlay an unexpired file, SAM prompts the operator to make a choice of either of the following:

  – Delete the unexpired file.

  – Cancel the job.

If the operator does not cancel the job, SAM deletes the labels of the unexpired file. This, in effect, removes the file (or just part of it) from the volume.

Processing on End of Volume

SAM recognizes an end-of-volume condition when the last (or only) extent on one volume has been processed and an extent on another volume is defined for the file. When this occurs, SAM passes control to your LABADDR routine (if one exists) to build and pass user-standard trailer labels. When all of these trailer labels are processed, SAM processes the labels on the next volume. When all labels on the new volume are processed, SAM continues to do I/O processing for your program. The processing of standard-user labels is discussed under "User-Standard Labels" on page 59.

Processing on End of File

This is different for output and input files.

Output File

When all records of an output file are written, your program must issue a CLOSE macro for the file. SAM then passes control to your LABADDR routine (if one exists) to build and pass user-standard trailer labels. When all of these labels are processed, SAM closes the file. The processing of standard-user labels is discussed under "User-Standard Labels" on page 59.

SAM signals an end-of-extent condition if the end of the file's last extent is reached before your program has issued a CLOSE.

Input File

An end-of-file condition exists when SAM either:

- Reached the end address of the file's last extent (as defined by an EXTENT statement for the file), or
- Found an end-of-file record.

SAM passes control to your end-of-file routine, whose name you must define in the EOFADDR operand of the DTFSD macro for the file.

User-Standard Labels

SAM assists you in the processing of user-standard labels.

A file may have up to eight user-header labels (UHL1, UHL2, and so on) and up to eight user-trailer labels (UTL1, UTL2, and so on). To process these labels, SAM requires you to:

- Code your own routine to build and write or to check these user-standard labels.

  In this routine, you cannot issue any macros that use the system's transient area. Typical examples of such macros are: OPEN, CLOSE, and CHKPT.

  To *build and write a label*, your routine must:

  1. Build the 80-byte label (leaving the first four bytes free – SAM inserts the correct UHLn or UTLn label identification).
  2. Pass this label to SAM when your routine returns control to SAM (by way of a LBRET macro).

  SAM writes your user-standard file labels onto the first track of the first (or only) extent of your file.

  To *check a label*, your routine must:

  1. Establish addressability of the label passed by SAM.
  2. Perform processing on the passed label as required.

If the labels of the file are to be checked against input from another file, that other file must be opened first.

Your program can update the label passed by SAM or can leave it unmodified. If the label is to be updated, your routine must:

a. Move the passed label to another location within your program.

b. Do the necessary processing on the label.

c. Pass the updated label to SAM when your routine returns control to SAM.

3. Return control to SAM by way of a LBRET macro.

- Define the name of the routine to SAM by specifying the LABADDR=name operand of the DTFSD macro.

Your routine receives control as follows:

- For checking user-standard labels

  SAM passes the labels to the routine, one at a time, until either of the following occurs:

  – The maximum number of labels has been read (and updated).
  – The routine indicates that it wants no more labels to be passed for checking.

- For writing user-standard labels

  To build the first (or only) user-standard label or, after such a label has already been written for a file, to build the next label.

Returning Control to SAM

The LBRET macro is issued in your subroutine when you have completed processing labels and wish to return control to IOCS. LBRET applies to subroutines that write or check standard disk labels. This macro has only one operand: one of the numerals 1, 2, or 3. Use one of these to specify which function you want SAM to perform. Specify:

LBRET 1
If, for the *checking of labels*, you do not want any of the remaining labels to be passed to your routine.

If, for the *writing of labels*, you want to stop further writing of labels before the maximum number of labels has been written.

LBRET 2
If, for the *checking of labels*, you want SAM to read and pass the next label (see also "Note" below).

If, for the *writing of labels*, you want SAM to return control to your routine after a previously built label has been written. However, if the maximum number of labels has already been written, SAM ends the processing of labels.

LBRET 3
This operand applies only to label checking. Specify 3 if you want SAM to update (rewrite) the previously read label and to pass to your routine the next label.

Note: If SAM finds an end-of-file record, the checking or updating of labels is stopped automatically.

Processing an Update File

Normally, a file processed by SAM is an input or an output file. However, with disk devices it is possible to use the same file for both input and output. Your program reads a logical record from the file and, after processing, writes the updated record back into the original location of the file. To do this, specify UPDATE=YES in the DTFSD macro for the file.

GET obtains records from the file in the usual way. After a record has been processed, the next PUT causes this record to be returned to its original location in the file.

A PUT for an update file must always be followed by a GET before another PUT is issued. The first PUT for the file must be preceded by a GET, of course, but if a record does not require updating, a subsequent PUT may be omitted. GET macros can be issued as many times in succession as desired.

In the (skeleton) update file example of Figure 35 on page 61, information is read, processed, and then written back into the same location on disk. Processing is done in the input area.

In your program, you issue a PUT when processing of a record is complete. The next GET (or CLOSE) then causes the actual data transfer to take place. Therefore, the input area must not be modified between a PUT and the next GET. If a work area is used for the file, PUT returns the records from the work area to the input area (or CI buffer) and then from this area to the file. For spanned records, you must have a work area, and this area must be sufficiently large to hold the entire spanned record.

```
                                                  Column 72
FILED     DTFSD       BLKSIZE=MAX,                    X
                      RECSIZE=100,                    X
                      IOREG=(6)                       X
                       ...
                      RECFORM=FIXBLK,                 X
                      UPDATE=YES
            ...
          OPEN        FILED
            ...
          GET         FILED
            ...
          PUT         FILED
            ...
```

Figure 35. Coding Example for Processing an Update File

Coding an Error-Processing Routine

As explained under “Error Handling (ERROPT, WLRERR, and ERREXT)” on page 44, there are two kinds of error-processing routines, identified by ERROPT and WLRERR in the DTFSD macro. ERROPT specifies how SAM is to handle read or write errors, WLRERR handles wrong-length errors.

In your error routine, you can code any kind of processing as long as you observe the following rules and restrictions:

- An IOCS macro other than ERET is not permitted.
- When an error condition occurs, register 1 contains the address of a two-word parameter list, which your routine can use as indicated:
  - First word – the address of the DTF table

    Your program can use this address to examine certain indicators in the CCB (the first 16 bytes of the DTF table). For a discussion of the CCB, see z/VSE System Macros Reference.
  - Second word – the address of the error block

    Your program can use this address to access the record for error processing (the content of the IOREG register, or of the work area if one is used, is unpredictable). If spanned records are processed, the address is that of the whole blocked or unblocked spanned record.
- In your routine, test the data transfer bit (bit 2 of CCB byte 2) to find out whether a non-recoverable I/O error has occurred. If the bit is:

  1
  The block in error has not been read or written.

  0
  The data was transferred, and your routine must access the error block to determine the action to be taken.
- At the end of error processing, your routine must return control to SAM in either of the following ways:
  - By the ERET macro

For an *input file*, this causes SAM to

- Skip the error block and to read the next block if you coded ERET SKIP.
- Ignore the error if you coded ERET IGNORE.
- Make another attempt to read the error block if you coded ERET RETRY.

For an *output file*, this causes SAM to

- Ignore the error condition if you coded ERET IGNORE or ERET SKIP.
- Make another attempt to write the block if you coded ERET RETRY.

– By branching to the address in register 14 when the error routine was entered.

For a read error, SAM skips the error block and makes the first logical record of the next block available for processing in your program. For a write error, SAM ignores the error and continues processing.

The two tables below summarize (1) the DTFSD error options and (2) the macro requests for various possible actions by SAM.

Desired Action	**Your DTF Specification**
Terminate the job:	Nothing
Skip the error record:	ERROPT=SKIP
Ignore the error record:	ERROPT=IGNORE
Process the error record in a user routine:	ERROPT=name
To process a wrong-length record error in a user routine:	WLRERR=name.

Desired Action	**Your Macro Specification**
After processing the record, return control to SAM and	
Skip the record (input only):	ERET SKIP request
Ignore the record:	ERET IGNORE request
Retry the record:	ERET RETRY request

Note: You cannot use the ERET RETRY option in your error routine when processing record length errors. For this condition, ERET RETRY causes the job to be canceled.

Wrong-Length Error

If a block read by SAM is shorter than the length specified in the BLKSIZE operand, the first halfword of the DTF table (CCB) gives the number of bytes left to be read (residual count). The size of the actual block is equal to the specified block size minus the residual count. A short block does not result in a wrong-length error condition for a file of variable-length or spanned records. If the block read by SAM is longer than the length specified in the BLKSIZE operand, the residual count is zero. The number of bytes read by SAM is equal to the length specified in the BLKSIZE operand. There is no way to compute the actual size of the block, and the remainder of the block is lost (truncated).

Undefined records are not checked for incorrect length. An undefined record is truncated if its length exceeds the length specified in the BLKSIZE operand.

Issue an ERET macro in your error routine (WLRERR=name) to skip or ignore a wrong-length error as described in the topic Coding an Error-Processing Routine. However, ERET RETRY is invalid for a WLRERR routine of a DTFSD file; it causes the job to be canceled.

For ERET IGNORE and WORKA=YES specified in DTFSD, do not destroy R0 in your error routine (WLRERR=name), as R0 may contain the work area address.

Other Errors

When a parity error occurs, the system tries to reread or rewrite the error block a certain number of times. If the attempts are unsuccessful, the job is canceled, except if the ERROPT operand in the DTFSD macro was specified.

If an error occurs when SAM rereads a block while updating a spanned record and neither WLRERR nor ERROPT were specified, the entire logical record is skipped. Likewise, if an error occurs when SAM rereads a block that contains the last segment of a blocked spanned record, the next entire logical record is skipped. If the DTFSD macro includes the operand WLRERR or ERROPT (or both), the error recovery procedure is the same as for non-spanned records.

A sequence error may occur if SAM searches for the first segment of a logical spanned record and fails to find it. If you specified WLRERR=name or ERROPT=name, the error recovery procedure is the same as for wrong-length record errors. If you specified neither of the two, SAM ignores the error and searches for the next first segment. Write errors are ignored.

Closing a File and Processing for End of Volume

This section discusses disk-file specific aspects of the closing of a file; it tells when a request for forcing end of volume is indicated and how you would code this request.

End of File

After a CLOSE for a file, no further requests can be issued for this file until it is reopened. A file defined by a DTFSD can be reopened successfully if your program:

1. Saved the DTFSD table before the file was first opened.
2. Restored the table between closing the file and reopening it again.

A CLOSE for an output file normally causes SAM to:

1. Write any outstanding data, for example the last block.
2. Write an EOF record and output trailer labels, if any.
3. Set a last-volume indication in the format-1 label of the file.

End of Volume for a Multi-Volume File

Both GET and PUT check for an end-of-volume condition. When such a condition occurs, SAM automatically performs volume switching.

You would force an end-of-volume condition for a multivolume file when your program has finished processing records for one volume and more records of the same logical file are to be processed for another volume. The FEOVD macro allows you to force end of volume before it actually occurs. If no extents are available on the new volume, or if the format-1 label is posted as the last volume of the file, control is passed to the EOF address specified in the DTFSD macro for the file. The remaining functions of SAM for an FEOVD macro differ for the various types of files:

- Input File

  SAM bypasses any remaining extents of the file on the current volume when your program issues the next GET for the file. SAM then opens the first extent on the next volume, and normal processing continues with the new volume.

- Output File on a CKD Disk

  SAM writes a short last block, with an end-of-volume indicator (end-of-file record containing a data length of 0) if necessary. In addition, SAM sets an end-of-extent indication in the DTF table. When your program issues the next PUT for the file, SAM:

  1. Bypasses all remaining extents on the current volume.

2. Opens the first extent on the next volume, after which normal processing continues with the new volume.

If the FEOVD macro is followed immediately by a CLOSE, then SAM:

1. Rewrites the end-of-volume marker as an end-of-file marker.
2. Closes the file in the usual way.

- Output File on an FBA Disk The processing is the same as for an output file on a CKD disk, except that SAM writes:

  1. The last data CI rather than a block.
  2. An FBA specific end-of-volume indicator (a CI of all zeros, referred to as SEOF), if there is room for it. If there is no room in the last CI to hold an SEOF, SAM considers the file to be delimited by end-of-last-extent.

Process a File Residing on an FBA Disk

Skip this section if your computer system does not include FBA disk devices.

Programs that use SAM to access data on a CKD disk can normally run unchanged with the data stored on an FBA disk. Exceptions are programs that are sensitive to I/O synchronization (such as error exits and logging). These programs have to be re-evaluated; they may have to be changed.

As we know, FBA disks use control intervals (CIs) as the unit of data transfer rather than blocks of data. In addition, the size of a CI may be different from the size of a logical block of a file. Therefore, issuing a macro that usually causes a block to be transferred need not cause an actual transfer of a CI. For instance, a WRITE macro to a file on an FBA disk causes a logical record to be moved from the output area to the CI buffer. When the CI buffer is filled, SAM transfers this buffer to the disk, asynchronously with the WRITE. SAM automatically reformats CIs into logical blocks and vice versa, as it automatically blocks and deblocks the records of a file.

If your program requires physical writes to be done when a formatted WRITE or a PUT is issued, your DTFSD macro must include the force-write (PWRITE=YES) operand. However, writing the content of a whole CI buffer for each record can slow down your system considerably.

A suitable size of a CI can affect overall throughput. For example, if this size is such that only one logical block (including control information) fits into the CI, the number of physical I/O operations needed to access a file is increased. If the size of the CI is large enough to contain two or more logical blocks, throughput is improved. Fewer physical I/O operations are necessary to access the same file.

Chapter 5. Processing a Tape File with SAM

This topic expands on the information contained in Chapter 3, “Defining and Processing a File with SAM,” on page 41; it gives tape specific hints for the use of SAM.

Defining a File

The following specifications in your DTFMT macro deserve some discussion: the block size, reading a tape backward, and the rewind option.

Block Size

The BLKSIZE operand of the DTFMT macro specifies the number of bytes to be transferred between the I/O area and the tape drive.

A block cannot be shorter than 12 bytes; a record of eleven bytes or less is treated as noise. A block cannot be longer than 32,767 bytes.

For output processing, the minimum block length is 18 bytes. If you define a block length of less than 18 bytes (but not less than 12), SAM:

- Writes the records padded up to a length of 18 bytes for fixed and variable-length records.
- Assumes BLKSIZE=18 for spanned records.

If a READ or WRITE macro for a work file specifies a length greater than the BLKSIZE value, the block to be read or written is truncated to fit into the I/O area.

For ASCII tapes, your specification for BLKSIZE must include the length of any existing block prefix or padding characters. Also, the defined length must be within the limits specified by American National Standards Institute, Inc. Therefore, if your specification for BLKSIZE is less than 18 bytes (for fixed-length records only) or greater than 2 048 bytes, the assembler generates an MNOTE.

Processing of Labels

To have SAM process labels, the DTFMT macro for the file must define the type of labels to be processed. This definition is one of the following:

FILABL=STD
For a file with IBM or user-standard labels. For a file with IBM standard labels and no user-standard labels, this is the only label-related operand that you must specify.

FILABL=NSTD
For a file with nonstandard labels.

FILABL=NO
For a file without labels.

If your file includes user-standard labels or has nonstandard labels, your program must include a label-processing routine. You define this routine to SAM in the DTFMT macro for your file by the operand LABADDR=name.

A tape file with standard labels must be defined to the system by way of a TLBL statement before SAM can open the file. For more information about this statement, see z/VSE System Control Statements.

User-standard labels (if any) always follow the IBM standard labels.

How SAM processes standard labels is presented separately for the various types of files.

Output File

Writing of IBM Standard Labels

An OPEN macro for the file when the tape is at its load point causes SAM to check the volume (VOL1) label.

For both EBCDIC and ASCII files, SAM supports HDR1, EOF1, and EOV1 labels, as well as HDR2, EOF2, and EOV2 labels. IOCS writes a HDR2 label behind the HDR1 label, an EOF2 label behind the EOF1 label, and an EOV2 label behind the EOV1 label.

If SAM finds that the first (or only) file on the tape has expired, SAM writes a new file header (HDR1 and HDR2) label for your output file.

If your program issues an OPEN for a file in the middle of a tape reel, it is your responsibility to position the tape past the tapemark at the end of the preceding file. You can use the MTC command to do this. SAM issues a message to the console if the tape is at a wrong position.

A CLOSE macro for the file causes SAM to write:

1. Any records of the file not yet written.
2. One tapemark.
3. An EOF1 and EOF2 label followed by two tapemarks.

SAM then performs the specified or defaulted rewind action.

End of volume may occur before your program has issued a CLOSE macro for the file. If the next I/O request for the file is a PUT, SAM starts end of volume processing. SAM then writes:

1. An unfilled (short) block of logical records followed by a tapemark.
2. An EOV1 and EOV2 label followed by a tapemark (two tapemarks for an ASCII file).

Writing of User-Standard Labels

For a non-ASCII file, up to eight user-standard labels may be written behind an IBM standard label:

UHL1 through UHL8 behind the HDR1 and HDR2 labels.
UTL1 through UTL8 behind the EOF1 and EOF2 labels.
UTL1 through UTL8 behind the EOV1 and EOV2 labels.

For an ASCII file, any number of user-standard labels may be written behind an IBM standard label.

To write user-standard labels, SAM requires you to:

- Code your own routine to build these labels.

  In your routine, you cannot issue any macros that use the system's Logical Transient Area. Typical examples of these macros are: OPEN, CLOSE, and CHKPT.

  The routine receives control when SAM has completed writing the required IBM standard label(s).

  Your routine must:

  1. Build the 80-byte label, including the label type identification (UHLn or UTLn) in the first four bytes.
  2. Pass the address of the label to SAM when the routine returns control to SAM.
  3. Return control to SAM by way of a LBRET macro.

  For details about coding the routine, refer to z/VSE System Macros Reference.

  After the last user-standard trailer label has been written, SAM does the required processing for end of file or end of volume.
- Define the name of the routine to SAM by specifying the LABADDR=name operand of the DTFMT macro.

Writing of Nonstandard Labels

An ASCII file cannot have nonstandard labels. To write nonstandard labels for a file, you must:

- Ensure that the tape is at the first label that is to be written. You can use the job control MTC command for this purpose.
- Write your own label build routine. In the routine, you have to write your own channel program and use physical IOCS macros to write the labels onto tape. In summary, this requires you to:

  1. Set up a command control block by issuing a CCB macro.
  2. Define a label input or output area.
  3. Write a channel program consisting of CCWs as required to transfer a label from this area.
  4. Issue an EXCP macro that refers to this CCB and the channel program.
  5. Return control to SAM by issuing the LBRET macro.

  The restrictions given above for a routine to build user-standard labels apply also to a routine for building nonstandard labels. For details about coding the routine, refer to z/VSE System Macros Reference.
- Define your routine to SAM by the LABADDR=name operand of the DTFMT macro for your file.

Input File

Checking of IBM Standard Labels

SAM expects the tape to be positioned at the load point when your file is opened. The first labels after the VOL1 label are the HDR1 and HDR2 labels. SAM locates the file to be accessed by the file sequence number in the TLBL job control statement.

Checking of Standard-User Labels

To have standard-user labels checked, you must:

- Code your own routine to do the actual checking.

  The routine receives control when SAM has completed checking the IBM standard label(s) as required.

  Your routine must:

  1. Establish addressability of the label passed by SAM.
  2. Perform processing on the passed label as required.

     If the labels of the file are to be checked against input from another file, that other file must be opened first.

     Your program can update the label passed by SAM or can leave it unmodified. If the label is to be updated, your routine must:

     a. Move the passed label to another location within your program.
     b. Do the necessary processing on the label.
     c. Pass the updated label to SAM when your routine returns control to SAM.
  3. Return control to SAM by way of a LBRET macro.

  For details about coding the routine, refer to z/VSE System Macros Reference.
- Define your routine to SAM by the LABADDR=name operand of the DTFMT macro for your file.

Checking of Nonstandard Labels

Usually, nonstandard header labels are written at the beginning of a file, and a tapemark follows the last one of these labels.

When a file with nonstandard labels is opened, the tape must be positioned at the first label that is to be processed. You can use the job control MTC statement to position the tape. To have nonstandard labels checked, SAM requires you to:

- Include, in your program, a routine that does the actual checking of the labels.

  SAM requires such a routine also if the nonstandard labels of a file need not be checked but were written without a tapemark at the end of the last (or only) header label. Your routine must indicate to SAM (by LBRET 2) the end of the file's label set.

  When the routine gets control during OPEN, SAM passes:

  – A label-identification code (the character O in register 0).
  – The representation of the logical unit being used, which is in the same format as bytes 6 and 7 of the CCB.

  When SAM finds a tapemark (which indicates end-of-file), SAM gives control to your routine. The routine then must:

  1. Find out whether an end-of-file or an end-of volume condition exists.
  2. Pass to SAM (in register 0) a code that indicates the finding of your routine.

  In your routine, you must write your own channel program and use physical IOCS macros to read the labels from (or write updated labels to) tape. In summary, this requires you to:

  1. Set up a command control block by issuing a CCB macro.
  2. Define a label input or output area.
  3. Write a channel program consisting of CCWs as required to transfer a label to or from this area.
  4. Issue an EXCP macro that refers to this CCB and the channel program.
  5. Return control to SAM by issuing the LBRET macro.

- Define the address of this routine in the LABADDR=name operand of the DTFMT macro for your file.

For details about coding the routine, refer to z/VSE System Macros Reference.

Unlabeled File

To understand this section, you may want to familiarize yourself with the volume layout of an unlabeled tape as shown by Figure 26 on page 34.

To process a file on an unlabeled tape, either specify FILABL=NO in the DTFMT macro for the file or omit the FILABL operand. SAM then assumes that the tape has no labels, regardless of what is stored on the tape. SAM merely reads tapemarks or data on input and writes tapemarks or data on output.

Input

An unlabeled file on a nine-track tape can be read backward. Unlabeled ASCII tapes without any leading tapemark can also be read backward.

Position the Tape

You must provide for the tape to be positioned. Else, the first GET for the file cannot make the first block of the file available for processing. You can position the tape to the correct location as described below:

- The first file on the tape is to be read:

  Specify REWIND=UNLOAD in the DTFMT macro for the file or omit the operand. This causes the tape to be rewound to the load point on OPEN.

  Alternatively, you can code a CNTRL REW macro or supply a job control MTC REW statement. You may, if this is practical, have your operator position the tape by way of an MTC command. The MTC statement (or command) is described in z/VSE System Control Statements.

- The file to be read is not the first one on the tape:

  Issue the required number of CNTRL macros with the FSF command. The macro causes the tape to be forward spaced to the next tapemark. The required number depends on:

1. The number of files on the tape preceding the file to be accessed.
2. The number of tapemarks between the files.

Another alternative is an MTC job control statement, as mentioned above for REWIND.

End of File

SAM assumes the end of the file when it reads the tapemark that follows the last data record. SAM passes control to the end-of-file routine whose address you specified in the EOFADDR=name operand of the DTFMT macro for the file. This routine should check whether an end-of-file or an end-of-volume condition exists. Consider requesting a reply from the operator on an end-of-volume inquiry.

- End of file:

  Your routine should handle this according to your end-of-data requirements.
- End of volume:

  Your program must issue an FEOV macro. This causes SAM to switch to the alternate drive for the file if ALT was specified on the applicable ASSGN statement in your job stream. If ALT was not specified, SAM issues a message to the operator.

Multiple Files on One Volume

If multiple files on the same volume are to be read in sequence, specify REWIND=NORWD in the DTFMT macro for each file. In that case, the tape is positioned correctly for the next file to be opened each time the input from one file is complete.

Output

When your program opens the file and the tape is positioned at its load point, SAM verifies that the first record on the tape is not a VOL1 label. If it finds a VOL1 label, SAM does not open an output file; it writes a message to the console instead. SAM waits for an operator decision to either ignore the label or to wait for a new tape reel to be mounted.

SAM writes tapemarks as follows:

1. At the beginning of the file if TPMARK=YES:

   One tapemark, starting at the location where the tape is positioned when your program issues the OPEN for the file.
2. At the end of the file:

   Two tapemarks behind the last block of data when your program issues a CLOSE macro after having processed all records of your file. However, SAM writes only one tapemark if either of the following occurs:

   - SAM finds the reflective marker at the end of the tape before the end of the output file.
   - Your program issued an FEOV macro for the file.

No tapemarks are written by SAM if you specify TPMARK=NO for the file.

Multi-File Volume

You may want two or more files to be written on the same tape without repositioning the tape. To do this, specify REWIND=NORWD in the DTFMT macro for each of the files. This causes the tape to stop after one of the tapemarks written behind the file, unless there are no tapemarks (TPMARK=NO), in which case it stops directly after the last block in the file.

Multi-Volume File

If the next I/O macro after the reflective marker at the end of the tape is a CLOSE, an end-of-file condition exists. If the next I/O macro is a PUT or an FEOV (forced end of volume), an end-of-volume condition

exists. In that case, SAM writes a tapemark and switches to the alternate drive. If no alternate drive has been specified, SAM requests the operator to mount a new volume. SAM positions the new tape at the load point and writes a tapemark if TPMARK=YES was specified in the DTFMT macro.

American National Standard Labels

SAM can process tape files that consist of ASCII records. These files may be unlabeled or labeled with American National Standard standard or user-standard labels; they may not have nonstandard labels. Following are some differences in label handling of which you should be aware:

- Additional volume labels (UVL1-UVL9), if present are ignored by SAM on input. They are not built by SAM on output.
- The default for the version number in the American National Standard file label is 00; the EBCDIC label version number defaults to 0.
- EOV labels on an EBCDIC tape file are followed by one tapemark; on an ASCII tape file these labels are followed by two tapemarks.

Return Control to SAM

In your routine for processing labels, issue a LBRET macro when the routine has completed processing a label and you wish to return control to SAM. LBRET is used in routines that write or check user-standard or nonstandard labels. The operand you use, the numeral 1 or 2, depends on the function to be performed:

- For the checking of user-standard labels

  SAM reads and passes the labels to your program, one at a time, until a tapemark is read or until you indicate that you do not want any more labels.

  Use:

  LBRET 2
  If you want to process the next label. If SAM encounters a tapemark, it automatically ends label processing.

  LBRET 1
  If you want to bypass any remaining labels.

- For the writing of user-standard labels

  In your routine, you build the labels one at a time and return to SAM. Use:

  LBRET 2
  If you want SAM to:

  1. Write the label that your routine has built.
  2. Return control to your label routine (at the address specified in LABADDR).

  LBRET 1
  If you want SAM to:

  1. Write the label that your routine has built.
  2. Do not return control to your label routine.

- For the checking or writing of nonstandard labels You must process all your nonstandard labels together. Use LBRET 2 after all label processing is complete and you want control to be returned to SAM.

End of Volume for a Multi-Volume File

Both GET and PUT check for an end-of-volume condition. When such a condition occurs, SAM automatically performs volume switching.

You can stop reading from or writing to a file on a volume before the normal end of volume occurs. You do this by coding the FEOV (force end of volume) macro in your program. The macro forces an end-of-volume condition before SAM encounters a tapemark or a reflective marker. It indicates that the processing of records with the current volume is finished and that more records for the same logical file are to be processed with another volume.

The FEOV macro performs the same functions that occur on a normal end-of-volume condition, except for the checking of trailer labels. If the file is:

- An input file

  SAM immediately rewinds the tape (as specified by REWIND) and provides for a volume change (as specified by ASSGN statements). SAM checks the standard header label on the new volume. It allows your program to check any user-standard header labels if LABADDR is specified.

  For a tape with nonstandard labels (FILABL=NSTD is specified), FEOV allows your program to check these labels, too.

- An output file

  SAM writes, onto the tape:

  1. A tapemark (two tapemarks for ASCII files).
  2. A standard trailer label and user-standard labels (if any).
  3. A tapemark.

  For a file with spanned records, a record may begin on a volume and the space left on the tape may be too small to contain the entire record. In this case, SAM forces an end-of-volume condition at the end of the last completed spanned record. The spanned record for which SAM could not find enough space is then written onto a new volume.

  Restriction: You cannot process a multivolume file with spanned records on an IBM 9346 tape device. A unit check with command reject indicated in the sense byte will occur.

  On the new volume, SAM writes the header label(s) in accordance with the DTFMT operands FILABL and LABADDR and the ASSGN statements.

  For a tape with nonstandard labels, SAM allows your program to write trailer labels on the completed volume and header labels on the new volume, if this is desired.

Tape File Extension

You can add additional data records to an already existing standard-labeled file if you:

1. Defined the file with the operands FILABL=STD and TYPEFLE=OUTPUT in the DTFMT macro.
2. Supply a TLBL job control statement for the file with the DISP operand as shown below:

   ```
   // TLBL MYTPEFL ...,DISP=OLD
   ```

   Instead of DISP=OLD, you may specify DISP=MOD. If the file exists, SAM sets up the tape for an extension of the file.

   For more details about the TLBL statement, refer to z/VSE System Control Statements.

For SAM, a file exists and can be extended if either of the following sets of requirements are fulfilled in addition:

- The TLBL statement **includes** a file sequence number and
  1. A file with this sequence number is on the volume.

2. The stored file identifier of this file matches the file identifier specified in the TLBL statement.

- The TLBL statement **does not include** a file sequence number and either of the following is true:
  - The tape is **at its load point** and the file identifier of the first file on this volume matches the file identifier specified in the TLBL statement.
  - The tape is **not at its load point** and the file identifier specified in the TLBL statement matches the file identifier of the succeeding file.

If the file to be extended does not exist, SAM issues a message and cancels the job.

SAM positions the tape to the end of the last data record in the file before extension starts.

A file to be extended may span more than one volume. Normally, you would want to extend such a file on its last volume. To start extension with a volume other than the first, supply the correct label information (such as file serial number and volume sequence number) in the TLBL statement. If there is a mismatch, SAM issues a message. SAM also issues a message if the file to be extended is expired. In that case, you must copy the entire file to get a new expiration date.

To avoid time-consuming tape skip operations for a file extension, always mount the last volume of a file's set of volumes.

Restriction: Tape file extension is not possible on an IBM 9346 tape device. SAM issues a message during Open and cancels the job.

Processing for IBM Standard Labels

For the File to be Extended

SAM compares all fields in the standard header label with the corresponding specifications in the // TLBL statement. SAM issues error messages if there are any discrepancies.

The standard header label (HDR1 and HDR2) of the file to be extended remains unchanged. However:

- Before the extension, the creation date in the standard header (HDR1 and HDR2) label as well as in the standard trailer (EOF1 or EOF2) label reflects the date when the file was created.
- After the extension, the creation date in the EOF1 (or EOF2) label reflects the date when the file was extended. The expiration date in the EOF1 (or EOF2) label remains unchanged.

For the File Behind the One to be Extended

SAM checks the expiration date of the file. SAM issues the FILE UNEXPIRED ... message if the file has not expired. On a reply of IGNORE, SAM overwrites the file.

Processing for User-Standard Labels

The user trailer labels on the existing file are not saved. Since SAM positions the tape to the end of the data, these labels will be overwritten when new data is written onto tape.

When the file is closed, your program's label routine (if present) receives control to build standard user trailer labels as required.

Coding an Error-Processing Routine

SAM can handle I/O errors and wrong-length record errors. Your program may include just one error-handling exit routine or one such routine for either of these types of error. The table below lists the possible error-processing actions and indicates (1) the DTFMT options and (2) the macro requests that must be specified to get these actions.

Desired Action	Your DTFMT Specification
Terminate the job:	Nothing
Skip the error record:	ERROPT=SKIP
Ignore the error record:	ERROPT=IGNORE
Process the error record:	ERROPT=name
Process a wrong-length record error:	WLRERR=name.

Desired Action	Your Macro Specification
After processing the record, return control to SAM and Skip the record (input only):	Either of the following • Execute BR 14. • ERET SKIP request.
Ignore the record:	ERET IGNORE request

The routine that you define in the ERROPT=name operand receives control when SAM encounters:

- An irrecoverable tape I/O error (ERREXT=YES is specified in the DTFMT macro for the file).

  ERREXT=YES must be specified for output files (TYPEFLE=OUTPUT).
- A tape read data check that the system was unable to correct (ERREXT=YES is not specified in the DTFMT macro for the file).

Your routine may perform any kind of error processing if it adheres to the following rules and restrictions:

- It may not issue a GET for the file.
- If it issues any other IOCS macro (except ERET when you have specified ERREXT=YES), the routine must save the contents of register 13 (with RDONLY) and register 14 before the macro is used. The routine must restore these contents again after having used the macro.
- If it issues IOCS macros which use the same read-only module that caused control to be passed to the error routine, the routine must provide another save area. One save area is used for the normal I/O, the second one for I/O operations in the error routine itself.

  Before your routine returns control to the module that entered the error routine, it must set register 13 to point to the save area originally specified for the task.
- If you specified ERREXT=YES, register 1 contains the address of a two-word parameter list when an error condition occurs. The contents of this two-word list are:
  - First word

    The address of the DTF table. Your error routine can use the address to test the data transfer bit (bit 2 of byte 2 of the table).

    If the bit is on, the block in error has not been read or written. If the bit is off, data was transferred and your routine must access the error block to determine what action is to be taken.
  - Second word

    The address of the block in error. Your error routine can use this address to access the data block for error processing. The content of the IOREG register or work area, if either is specified, is unusable. When spanned records are processed, the word contains the address of the whole spanned record, blocked or unblocked.

  If you did not specify ERREXT=YES and an error occurs, then register 1 contains the address of the block in error. Your routine should use this address to access the error block for error processing.

Note: For ASCII tapes, the address in register 1 points to the first logical record following the block prefix.

- At the end of error processing, your routine must return control to SAM. Code either a branch to the address in register 14 or, for an input file, the ERET macro with the IGNORE or SKIP option (if you have specified ERREXT=YES). The RETRY option of the ERET macro is not valid for tape files; if used, the RETRY option causes your job to be canceled.

Wrong-Length Error Processing Considerations

A block read by SAM may be shorter than the length specified in the BLKSIZE operand. If so, the first two bytes of the DTF table contain the residual count, the number of bytes left to be read. The actual size of the block therefore is equal to the value specified for BLKSIZE, minus the residual count.

A block read by SAM may be longer than the length specified in the BLKSIZE operand. In that case, the residual count is zero, and there is no way to compute the actual size of the block. The number of bytes transferred is equal to the value specified for BLKSIZE, and the remainder of the block is lost.

For *fixed-length unblocked* records, a wrong-length record error condition exists when the length of the record read is not the same as the value specified for BLKSIZE.

For *EBCDIC fixed-length blocked* records, the record length is considered incorrect if the block being read is not a multiple of the logical record length as specified in the RECIZE operand. This permits reading of short blocks of logical records without a wrong-length record indication.

For *EBCDIC variable-length* records, blocked and unblocked, the record length is considered incorrect if the length of the block on tape is not the same as the block length specified in the four-byte block length field. The residual count can be obtained by addressing the halfword in the DTF table at filename+98.

For *ASCII variable-length* records, blocked and unblocked, SAM checks the length of each block if LENCHK=YES is specified. The length of a block is considered incorrect if the tape record is not the same as the block length specified in the four-byte block prefix. In this case, the WLR bit in the DTF table (bit 1 of byte 5) is set off.

For *undefined* records, a wrong-length record is indicated if the record read is longer than the value specified for BLKSIZE.

Other Error-Processing Considerations

While reading a block of records, SAM may detect a **parity error**. SAM then backspaces and rereads the block a certain number of times before it considers this block as an error block.

An output parity error is considered to be an error block if the error exists after SAM has attempted a certain number of times to forward erase and to write the block. Your error processing routine should treat the device as inoperative; it should not attempt further output to this device. Any subsequent attempt to return to SAM causes the job to be canceled.

A **sequence error** may occur while SAM is searching for a first segment of a logical spanned record and fails to find it. If you specified WLRERR=name or ERROPT=name, the error recovery procedure is the same as for wrong-length record errors. If you specified neither, SAM ignores the sequence error and searches for the next first segment.

Non-Data Operations

Note: If your program issues a CLOSE for an unopened tape-input file, SAM performs the option specified in the REWIND operand of the DTFMT. For an unopened tape-output file, a CLOSE does not cause a tapemark or any labels to be written; also, no REWIND option is performed.

By way of the CNTRL macro, your program can control a number of tape-handling operations that are not concerned with reading or writing data. As an operand of the macro, you specify a function code, a mnemonic operation request. You can select the desired operation from a set of codes for function categories as follows:

- Rewinding a tape to the load point
- Moving a tape to a certain position.
- Spacing forward or backward by one logical record.
- Writing a tapemark.
- Erasing a portion of the tape.
- Synchronizing the hardware buffer (if you have a buffered tape unit) with your program.

For a list of all possible function codes, see the description of the CNTRL macro in z/VSE System Macros Reference. Some of the function categories are further discussed below.

Rewind and Tape-Movement Functions

Your program can use the function codes of these categories before a tape file is opened. This allows your program to position the tape at the desired location for opening a file. In other words, your program can cause the tape:

- To be positioned to a file located in the middle of a multifile reel.
- To be rewound even if NORWD was specified in the REWIND operand of the DTFMT macro for the file.

Note: If your program is self-relocating, you must open the file before issuing any commands for it.

The tape movement functions apply only to input files. They always start at an inter-block gap. For the use of these functions, consider the following:

- The FSF (forward space file) or BSF (backward space file) function permits you to skip to the end of the file (identified by a tapemark).
- The FSR (forward space record) or BSR (backward space record) function permits you to skip over a block of data, from one inter-block gap to the next.
- If you have blocked input records and you do not want to process the remaining records of a block, nor any of the succeeding blocks, issue a RELSE macro before the CNTRL macro. The next GET then makes the first record of the new block available for processing. If the CNTRL macro (with FSR, for example) is issued without a preceding RELSE, the tape is advanced; the next GET makes the next record in the old block available for processing.
- For any I/O area combination, except one I/O area and no work area, SAM always reads one tape block ahead of the one that is being processed. Thus, the next block after the current one is in storage ready for processing.

  If a CNTRL FSR is given, the second block beyond the present one is skipped without being read into storage.
- If FSR or BSR is used, SAM does not update the block count. Moreover, SAM cannot sense tapemarks. Therefore, SAM does not perform the usual end-of-file or end-of-volume functions if such an end condition occurs.

Spacing Over a Logical Record

The tape spacing functions FSL (forward space logical) or BSL (backward space logical) apply only to input files with spanned records. Consider the following when you plan to use FSL or BSL:

- Logical record spacing is ignored if it immediately follows a RELSE macro.
- Forward and backward spacing refer to the actual direction of the spacing. For example, if BSL is specified for an input file that is being read backwards, SAM skips one logical record.
- If an end-of-file, end-of-volume, or an error condition occurs while an FSL or BSL spacing function is being executed, the condition is handled as if it occurred during a normal GET operation.

Synchronizing the Hardware Buffer

The function code in this category (SYN) applies to output to a buffered tape unit that operates in buffered mode. For example, the macro

```
        CNTRL TAPEOUT,SYN
```

causes SAM to write to the file TAPEOUT any of your program's output that is still in the buffer of the involved tape unit. Your program receives control again when this write operation is complete.

User Interface for Tape OPEN, CLOSE, and End-of-Volume

Note: This interface was implemented primarily for IBM supplied tape-managing systems; if you want to use it, you must provide your own tape managing routine.

With this interface, a user program (normally a tape management system) can get control:

1. Before OPEN.
2. Before switching to an alternate volume.
3. Before CLOSE.
4. After CLOSE.

If the tape management system is to get control at these points, it must provide an exit routine as phase $IJJTXIT and load this phase into the SVA. If the phase $IJJTXIT does not exist, processing continues without any error indication.

If the SVA includes a routine of your own as phase $IJJTXIT, this routine can do any processing, except issuing an IOCS macro for the file. If the routine does, SAM cancels the job. Return to SAM by a branch to the address in register 14.

SAM passes control to the routine with registers set as follows:

Register 0
A function code indicating the point of processing:

00 =
Before OPEN

04 =
Before CLOSE

08 =
After CLOSE

0C =
On end of volume, before alternate-tape processing

Register 1
A pointer to the DTF table (DTFMT or DTFPH)

Register 13
Address of a save area

Register 14
Return address

Register 15
Entry point of $IJJTXIT

On return from $IJJTXIT, registers 0 through 14 are restored; register 15 has to contain one of the following return codes:

0 =
Always, except for certain end-of-volume (EOV) processing.

4 =

Applies to EOV processing – Normal alternate tape assignment processing is to be skipped. The tape management system has made the alternate assignment and has ensured that the correct volume is mounted.

Access-Protection for an ASCII Tape

In accordance with the ANSI Standard, Level 3, the system denies access to an ASCII file that is access-protected.

The VSE system as shipped includes two phases named $IJJTSEC and $$BOMTAC. These phases check the accessibility byte in the VOL1 and HDR1, HDR2 labels. If this byte is set to X'40', X'00', or C'0', SAM cancels the job. You can replace these phases by one of your own that meets the requirements at your location. To do this, change the A-type macro IJJT$SEC or BOMTAC accordingly and reassemble and link the macro. You may use the sample job stream below:

```
// JOB IJJTSEC
// OPTION CATAL,NODECK
// EXEC ASMA90....
   COPY IJJT$SEC|BOMTAC
   END
/*
// EXEC LNKEDT,PARM='MSHP'
/&
```

The statement //EXEC ASMA90.... calls the High Level Assembler. Refer to the topic Using a Macro in a Program for further details.

Chapter 6. Processing a Unit Record File with SAM

A unit record file can use a wide variety of I/O devices. These range from punched card through printer to the console. For some of these, each record is complete on one unit of information storage. For other files, such as printer or console, a unit is the line of print or display.

Because of this variety, unit record programming is quite device dependent. Different DTFxx macros are needed to define files for different types of unit record I/O devices. This topic discusses unit record files on device types as follows:

- Punched Card I/O
- Printer
- Console

Processing a Punched Card File

To process a punched card file, your program must define the file by a DTFCD macro and a CDMOD (logic module generation) macro.

Punched card equipment normally performs only one function, for example reading cards or punching cards. Certain types of punched card equipment, so called card read punch machines, can perform both of these functions in a single path.

The IBM 3525 can also print on the cards of a file. Logically, however, data to be printed is a separate file; you must define this file in your program by a DTFPR macro and must also code a PRMOD macro. The section “Programming for Associated Files” on page 79 provides hints for using the IBM 3525 to perform multiple functions on a file in one pass.

A card file on the IBM 3525 can be both read and punched in one run. To accomplish this, your program must define the file as combined. Coding considerations for a combined file are given under “Updating a Record” on page 81.

The IBM 3505 offers support for an additional kind of application: this card reader can be equipped with the optical mark reader feature. The feature allows up to 40 columns of marked data to be read. Hints for dealing with this data are given under “Optical-Mark-Read and Read-Column-Eliminate Modes” on page 81.

Programming for Associated Files

This section applies to a card file on an IBM 3525.

If only one function is to be performed, specify one of the following DTFxx macros as listed below:

FUNC=R
For reading – in the DTFCD macro

FUNC=P
For punching – in the DTFCD macro

FUNC=W
For printing – in the DTFPR macro

FUNC=I
For punching and interpreting – in the DTFCD macro.

If two or more card functions are to be performed on the cards of a file, define this file as a set of *associated* files. All combinations of the three card functions (read, punch, and print) are possible; you need to define each of the desired functions by a separate DTFxx macro: DTFCD for the read and punch functions and DTFPR for the print function. Your file definitions are associated by your specifications in the ASOCFLE operands.

For an associated file, you can define only one output area (with or without a work area). You can define the same output area for both files or a separate output area for each of the files. For example, if the same information is to be punched and printed, it is of advantage to use only one output area for both files.

Note: If you use associated files and ASSGN ...,IGN, the logical units for both files must be assigned IGN.

Read-Punch Associated Files

Specify FUNC=RP in the DTFCD macros for the two files.

For each card, SAM requires a GET for the read file and a PUT for the punch file. If no punch operation is desired for a card, your program must fill the output area with blanks before it issues the PUT macro.

If you specified CTLCHR=YES or CTLCHR=ASA for the punch file, the control character must always be present in the first byte of the output (or work) area. The data area following the control character may be filled with blanks. If you use the CNTRL macro instead, issue this macro before the PUT for the punch file. By issuing this CNTRL as soon as possible after the GET for a card, you can improve card throughput.

Read-Print Associated Files

Specify FUNC=RW in the DTFCD macros for the two files.

For each card, SAM requires a GET for the read file. A PUT for the print file needs to be issued only when a line is to be printed. However, this PUT causes a new card to be fed. Therefore, consider to fill the output (work) area with blanks and then issue the PUT macro.

If you do not issue a PUT for the print file, overlapped processing cannot take place. This may slow down card throughput.

Read-Punch-Print Associated Files

Specify FUNC=RPW in the DTFxx macros for the files.

For each card, SAM requires a GET for the read file and a PUT for the punch file. Regarding PUT macros for the print file, see “Read-Print Associated Files” on page 80, above. Refer to “Read-Punch-Print Associated Files” on page 80 for the effect of the CTLCHR operand in the DTFCD macro and a CNTRL for the punch file.

For *associated files*, GET, CNTRL, and PUT macros must be used in the sequence as shown in Table 6 on page 80. For example, to process a card of a read-punch associated file requires this sequence:

1. A GET macro for the file defined in the read DTFCD.
2. A CNTRL macro (if desired) for the file defined by the punch DTFCD.
3. A PUT macro for the file defined by the punch DTFCD.

A sequence of GET and PUT macros other than as shown cause an abnormal end with an ILLEGAL SUPERVISOR CALL message. The use of CNTRL in a sequence other than as shown causes unpredictable results.

Table 6. Sequence of GET/CNTRL/PUT Macros for Associated Files

Each macro sequence processes one card of the involved file of cards.			
Function	**Sequence of macros**	**For the file defined by:**	**FUNC=**
Read/Punch	GET CNTRL * PUT	DTFCD (read file)DTFCD (punch file)DTFCD (punch file)	RP
Read/Punch/Print	GET CNTRL * PUT PUT **	DTFCD (read file)DTFCD (punch file)DTFCD (punch file)DTFPR	RPW

Table 6. Sequence of GET/CNTRL/PUT Macros for Associated Files (continued)			
Each macro sequence processes one card of the involved file of cards.			
Function	**Sequence of macros**	**For the file defined by:**	**FUNC=**
Read/Print	GET CNTRL * PUT **	DTFCDDTFCDDTFPR	RW
Punch/Print	CNTRL * PUT PUT **	DTFCDDTFCDDTFPR	PW

*****
Optional. If used, however, the sequence is as shown.

******
Optional if you do not want to print on the card; if used, however, the sequence is as shown.

Updating a Record

Some IBM card devices, the IBM 3525 for example, can read a card and allow additional information to be punched back into that card. A card file to be processed in this way is called a combined file.

In the DTFCD for a combined file, you must specify TYPEFLE=CMBND. For an IBM 2540 with the punch-feed-read feature, the file to be updated must be in the punch feed.

An example of a combined card file is given in Figure 36 on page 81.

```
                                                  Column 72
FILEC     DTFCD       IOAREA1=AREA,                   X
                      IOAREA2=AREA2,                  X
                      DEVADDR=SYS005,                 X
                      RECFORM=FIXUNB,                 X
                      TYPEFLE=CMBND
           ...
          GET         FILEC
           ...
          PUT         FILEC
            ...
```

Figure 36. Example of a Combined File

In the combined card file example of Figure 36 on page 81, data is punched into the same card which was read. Information from each card is read and processed. The result of this processing is then punched into the same card to produce an updated record.

In your program, you can specify just one I/O area (by the IOAREA1 operand) for both the input and output of a card record. You can specify a second I/O area (with the IOAREA2 operand), if this is desirable.

A PUT for a combined file must always be followed by a GET before another PUT is issued. The first PUT must, of course, also be preceded by a GET. GET macros can be issued as many times in succession as desired, except when you use an IBM 2540. For a file on the IBM 2540 (with the punch-feed-read feature), your program must issue a PUT macro for each card.

The operator must run out the 2540 punch following a punch-feed-read job.

Optical-Mark-Read and Read-Column-Eliminate Modes

This section deals with input files to be processed in either of the following modes:

- Optical-mark-read (OMR) mode on an IBM 3505.
- Read-column-eliminate (RCE) mode on an IBM 3505 or 3525.

Either mode requires a format-descriptor card; OMR mode requires additional considerations regarding the data records.

In the DTFCD macro for the file, specify MODE=O for OMR mode and MODE=R for RCE mode.

Format Descriptor Card

The card defines the columns to be read or eliminated. When it finds this card, SAM builds an 80-byte record which relates to the specified format. If no format descriptor card is present, SAM issues a message to the operator and cancels the job.

The format descriptor card is to be coded as follows:

```
FORMAT  (n1,n2)
```

or

```
FORMAT  (n1,n2),(n3,n4),...
```

Rules for coding:

- The operation, FORMAT, is to be punched into columns 2 through 7.
- Operands begin in column 9 and may continue through column 71; they must be separated by commas as shown.
- Continuation cards can be specified by punching an X in column 72; coding on the continuation card must begin in column 16.
- The values for n1, n2, and n3 must be as follows:

  ```
  1 =< n1 =< n2 < n3 =< n4 ... =< 80
  ```

- For OMR, the value of n3 minus n2 must be greater than or equal to 2.
- For MODE=O, n1 indicates the first column, and n2 indicates the last column to be read in OMR mode. Only every other column between n1 and n2 can be read in OMR mode; therefore, n1 and n2 must both have either odd or even values.

  For example, if you want to read columns 1, 3, 5, 7, 9, 70, 72, 74, 76, 78, and 80 in OMR mode, you would use the following format descriptor card:

  ```
  FORMAT  (1,9),(70,80)
  ```

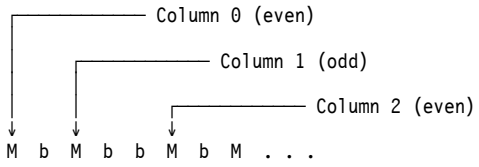
- For MODE=R, n1 indicates the first column *not* to be read, and n2 indicates the last column *not* to be read. Assume that all card columns except 20 through 30 and 52 through 76 are to be read. You would have to code a format descriptor card as follows:

  ```
  FORMAT  (20,30),(52,76)
  ```

OMR Data Card

The following rules apply to the coding of an input card to be read in OMR mode:

- A mark character (character to be read optically) must be separated from another mark character or a punched character by at least one column that contains neither marks nor punches.
- A mark character in an odd column must be separated from mark characters in adjacent even columns by at least two columns that contain neither marks nor punches. In the example below M indicates a marked column, and b indicates a blank column:



- A mark character must be separated from any columns containing punched holes (in the example indicated by 'H') by at least one column that contains neither marks nor punches:

```
  H  b  M  b  H  H  H
```

Although OMR data is contained in alternating card columns, this data is compressed into contiguous bytes in the I/O area. The relationship of the data on card columns to the location of the data in storage is as follows:

1. If column n does not contain OMR data, the data content of column n+1 represents the contiguous byte in virtual storage which follows the column n data byte.
2. If column n does contain OMR data, the data content of column n+2 represents the contiguous byte in virtual storage which follows the column n data byte. The data content of column n+1 is not read into virtual storage.
3. The data content of column 1 always represents the first data byte in virtual storage.

Figure 37 on page 83 shows how these rules apply to the data card and its format descriptor card; it shows the record that results from reading the data card.

Format Descriptor Card:

```
   FORMAT (4,6),(9,11)
```

Card Columns	1	2	3	4	5	6	7	8	9	10
Card Data	P1	P2	b	M4	b	M6	b	b	M9	b
Format Data	b	b	b	F4	-	F6	-	b	F9	-
			Switch from punch to mark					Switch from even marks to odd		
Channel Data	P1	P2	b	M4	M6	b	M9	M11	P13	P14
Card Columns	11	12	13	14	15	16	17	18	19	20
Card Data	M11	b	P13	P14	P15	P16	P17	P18	P19	P20
Format Data	F11	-	b	b	b	b	b	b	b	b
			Switch from mark to punch							
Channel Data	P1	P2	b	M4	M6	b	M9	M11	P13	P14

```
   b =  Must have neither hole nor mark data; is X'40'
   - =  May be character or blank
  Px =  Punched data in column x
  Mx =  Mark data in column x
  Fx =  Format data for column x
```

Figure 37. OMR Data and Format-Descriptor Example

Weak Mark

When SAM finds a weak mark or a poor erasure in a column, the column's data is replaced with:

X'3F'
When SAM reads in EBCDIC mode.

X'3F3F'
When SAM reads in column binary mode.

If X'3F' is placed in the data, an X'3F' is also placed in byte 80 of the I/O area when reading in EBCDIC mode, or in byte 160 when reading in column binary mode. This indicates an OMR reading error. By checking this byte, your program can determine whether an OMR-read error occurred on the card.

However, if the I/O area length is less than 80 for EBCDIC mode, or less than 160 for column binary mode, the X'3F' is not placed in virtual storage. To determine whether a reading error occurred, your program must check each OMR byte for an X'3F'.

End-of-File Handling

On end of file, SAM automatically transfers control to your end-of-file routine, which you define by way of the EOFADDR operand. This operand is required for input and combined files. In the end-of-file routine your program can perform any operations required for the end of the file. Generally, the routine issues a CLOSE instruction for the file.

SAM detects an end-of-file condition in the card reader by recognizing the characters /* punched in card columns 1 and 2 (column 3 must be blank).

If the system input is together with the job input and two I/O areas are used on the DTFCD, an extra blank card is required after end of file.

When MODE=O or MODE=R was specified in the DTFCD macro, a blank card must follow the card which causes your program to close the file.

If you have associated files on your IBM 3525, your program must issue a CLOSE for the two associated files without an intervening I/O operation. Reopening one associated file requires reopening also the other(s). The card movement caused by issuing CLOSE for a file on the IBM 3525 is as follows:

File Type	Feed Caused by CLOSE for:
Read	Read (see 1 below)
Punch	Punch
Print	Print
Read/Print	Print (see 1 below)
Read/Punch/Print	Print (see 2 below)
Read/Punch	Punch (see 2 below)
Punch/Print	Print
Punch/Interpret	Punch

Note:

1. A card feed is executed only if R has been specified in the DTFCD MODE operand. Programs using read-column- eliminate mode must detect an end-of-file condition themselves.
2. Delimiter cards cannot be punched or printed in these files. CLOSE always issues a feed command.

Error Handling

The ERROPT operand of the DTFCD macro specifies the error option used for an input or output file.

Hints for Programming

Card Feeding on an IBM 2540

When the machine is used for a card input, each GET macro normally reads the record from a card in the read feed. However, if (1) your machine has the punch-feed-read feature and (2) TYPEFLE=CMBND is specified in the DTFCD macro, each GET reads the record from the card at the punch-feed-read station. Your program can update this record with additional data. Such data is punched back into the same card when the card passes the punch station and a PUT macro was issued.

Printing on the IBM 3525

If your program opens a print-only file, SAM causes the first card to be fed to ensure that a card is at the print station.

If associated files are opened for a file of cards, all of the associated files must be opened before a GET or a PUT is issued for any of them.

For a machine with a 2-line print feature, output is automatically printed on lines 1 and 3. When automatic line positioning is used for a print-only file on a 2-line printer, then one PUT macro causes line 3 to be printed and the other PUT causes a new card to be fed and printing on line 1 to be started.

On a machine with a multiline print feature, card feeding is caused by the PUT macro that follows the PUT for printing on line 25. This PUT also starts the printing on line 1 of the next card.

If you want to control line positioning, specify the CONTROL operand or the CTLCHR operand in the DTFPR macro Neither of these operands is valid for card feeding when it is specified for a printer file associated with a punch file. In your program, you must provide for line spacing and skipping during printing. If you specify CTLCHR=YES, your program must also control card feeding.

The following restrictions apply to user-controlled line positioning:

- Any attempt to print on lines other than lines 1 or 3 with a 2-line print feature results in a command reject. Otherwise, 2-line print support is identical to the multiline print support.
- A space-after-printing command for line 25 results in positioning on line 1 of the next card.
- Any attempt to print and suppress spacing results in a command reject.
- Any skip command to a channel number less than or equal to the present channel position results in line positioning at that channel position on the next card.
- If CONTROL or CTLCHR is specified, FUNC is ignored for a 2-line print support.

Additional hints for programming the printing of cards on an IBM 3525 are included in the sections “Non-Data Device Operations” on page 85 and Chapter 6, “Processing a Unit Record File with SAM,” on page 79.

Non-Data Device Operations

Some device functions, such as output stacker selection or line spacing and skipping, if card printing is used, can be controlled either by specified control characters in the data records or by the CNTRL macro. Either method, but not both, may be used for a file.

Stacker selection can, in addition, be controlled by way of the SSELECT operand of the DTFCD macro for your file. If you omit this operand and use neither control characters nor the CNTRL macro, all cards go into the normal read or punch stacker. In your program, you can use the CNTRL macro to temporarily override the stacker selected by an SSELECT specification.

The CNTRL Macro

You cannot use the macro for:

- An input file with two I/O areas (the IOAREA2 operand is specified).
- A printer or punch file if the records of the file include control characters and you specified the operand CTLCHR in the DTFxx macro.

If you use the CNTRL macro in your program, include the operand CONTROL in your DTFCD or DTFPR macro and omit the operand CTLCHR. If your records include control characters although CONTROL is specified, SAM treats the control characters as data.

Do not specify the CONTROL operand in the DTFCD macro for an input file that is associated with a punch file on the IBM 3525 (FUNC=RP or FUNC=RPW is specified in the DTFCD macro). However, you may, if this is desirable, specify CONTROL in the DTFCD macro for the associated punch file.

The CNTRL macro usually requires two or three operands:

- The name of the file specified in the DTFxx macro; it can be specified as a symbol or in register notation.
- The mnemonic code for the command to be performed. This must be one of a set of codes applicable to the device.
- A stacker-select or a print-control value.

For a list of applicable command codes and stacker-select or print-control values, see the CNTRL macro in z/VSE System Macros Reference.

2540 Card Read Punch Codes

You can use the CNTRL macro with code PS to select a card into a different stacker pocket. Specify this pocket as the third operand, n1. The possible selections are shown below; they are the codes that you can specify also in the SSELECT operand.

Feed	Stacker	Value of n1
Read	R1	1
Read	R2	2
Read	RP3	3
Punch	P1	1
Punch	P2	2
Punch	RP3	3

To have a certain card stacked into a certain pocket, issue the CNTRL macro as follows:

- For an *input file* – After the GET macro for this card.

  If your program:

  1. Requires operator intervention – to correct a card out of sequence, for example, and
  2. Has specified CONTROL=YES in the CDMOD macro, then

  code a CNTRL macro before the operator intervention is requested. This assures that any command issued to your IBM 2540 after the operator intervention is not rejected as invalid.
- For an *output file* – Before the PUT for this card. However, CNTRL does not have to precede every PUT.

3505 Card Reader and 3525 Card Punch Codes

Cards read on the IBM 3505 or punched on the IBM 3525 are directed to the stacker specified in the SSELECT operand of the DTFCD macro. If SSELECT is omitted and no CNTRL is issued in the program, SAM directs the cards to stacker 1. However, if you code a CNTRL macro to have certain cards stacked into stacker 2 or 3, then stacker 1 must be selected explicitly. The CNTRL macro overrides the stacker selection specified in the SSELECT operand explicitly or by default.

3525 Card Print Codes

The CNTRL macro can control spacing and skipping to a specific line on a card on the IBM 3525 with the card print feature. The command code SP requests the IBM 3525 to space one, two, or three lines on a card; SK requests a skip to a channel (1 through 12) on a card.

The print channels correspond to specific rows on a card as shown in Figure 38 on page 87.

```
Line Number    Corresponding Channel

     1 ..................  1
     2
     3 ..................  2
     4
     5 ..................  3
     6
     7 ..................  4
     8
     9 ..................  5
    10
    11 ..................  6
    12
    13 ..................  7
    14
    15 ..................  8
    16
    17 ..................  9  (overflow)
    18
    19 ..................  10
    20
    21 ..................  11
    22
    23 ..................  12  (overflow)
    24
    25
```

Figure 38. Print Channel to Card Row Correspondence on an IBM 3525

Control Characters

When control characters are used in the records of your file, specify the CTLCHR operand in the DTFxx macro. Every record of the file must have a control character in its first character position if the file has fixed-length or undefined records. This control character must be the first character following the record-length field in a variable-length record. The BLKSIZE specification for the output area must include the byte for the control character. If undefined records are specified, the RECSIZE specification must include this byte.

In case of a PUT request, the control character in the data record determines the command code (byte) of the CCW that is built by SAM. The control character is used as follows:

If CTLCHR=YES
SAM uses the control character as the command code. Therefore, the supplied code must be valid for the device.

If CTLCHR=ASA
SAM translates the control character into the corresponding command code for the device.

To effect a space or a skip without printing, your program must supply the corresponding control character in the first character position of the output area for the file. The remainder of the area must be all blanks (X'40').

If RECFORM=UNDEF, the length of the record must be at least 2; if RECFORM=VARUNB, it must be at least 6.

To print on cards on the IBM 3525, you must use a space-1 control character (a blank) to print on the first line of a card.

The first character after the control character in the output data becomes the first character punched or printed.

For a complete list of all control characters, see z/VSE System Macros Reference.

Processing a Printer File

A printer file must be defined with the DTFPR macro and, possibly, the PRMOD macro. A PRMOD macro is not required for an IBM PRT1, 3800, 4248, or 6262 printer.

Associated File on an IBM 3525

For a discussion of associated files, see “Programming for Associated Files” on page 79.

To define an associated file for the IBM 3525, use the ASOCFLE operand together with the FUNC operand:

ASOCFLE Operand
It specifies the name of an associated read or punch file; it causes SAM to check the macro sequence in your program for each of the associated files. One name must be specified per DTFxx macro for an associated file.

FUNC Operand
It specifies the type of file to be processed on the IBM 3525. You can specify one of the following (followed by T if your IBM 3525 has a 2-line print feature):

RW	To indicate "read and print."
RPW	To indicate "read, punch and print."
PW	To indicate "punch and print."

A code specified for an associated printer file must be specified also for the associated card file(s). Do not use T (two-line) for the associated card file(s), it is valid only for your printer file.

T is ignored if your DTFPR includes CONTROL or CTLCHR.

Printer Overflow

The PRTOV (printer overflow) macro is used to specify the operation to be performed when a page overflow occurs. An overflow is caused by spacing into or beyond channel 9 or 12 in the forms control buffer (FCB). To use the PRTOV macro, include the PRINTOV=YES operand in your DTFPR macro.

SAM performs the action requested by the PRTOV macro when an overflow condition (channel 9 or 12) occurs. This is either a skip to channel 1 or a branch to the specified routine. However, SAM cannot detect an overflow condition during a skip operation.

Issue a PRTOV macro after any macro that causes carriage movement (PUT or immediate CNTRL) and before you issue the next CNTRL or PUT. This ensures that your overflow option is performed at the correct time.

If your program includes an overflow routine, return control from this routine by a branch to the address in register 14.

For output to an IBM 3525 with the two-line print feature, SAM ignores a PRTOV macro.

On an IBM 3525 with the multiline print feature, an overflow condition from channel 9 or 12 causes either of the following:

- A transfer of control to the overflow routine specified in the PRTOV macro.

  Note: PRTOV without a routine name is invalid for an associated file on the IBM 3525.
- A skip to channel 1 to begin printing on the next card for a print-only file.

Printer Controls

Line spacing or skipping for a printer can be controlled by either of these methods:

- Specify the CTLCHR operand in the DTFPR macro for the file and include a control character in each of your data records.
- Specify the CONTROL operand in the DTFPR macro for the file and use the CNTRL macro.

You can use either method, but not both, for the same file.

If your DTFPR does not include the CTLCHR operand, a PUT macro for the printer file causes the printer to space automatically by one line. If this meets your requirements, there is no need to issue a CNTRL macro or to specify a control character in each record.

How to use of the CNTRL macro is described under "CNTRL Macro" on page 89.

Control Characters

To use control characters, specify CTLCHR=YES or CTLCHR=ASA in your DTFPR. Every record of your file must have a control character either in the first character position of each fixed-length or undefined record, or in the first character position after the record-length field of a variable-length record. The value you specify for BLKSIZE must include the byte for the control character. If your file has undefined records, your value for RECSIZE must include this byte too. For maximum and default output area sizes for the various types of printers, see the description of the DTFPR macro in z/VSE System Macros Reference.

Note: Printing without spacing can be done only with the CTLCHR operand.

In case of a print (PUT) request, the control character in the data record determines the command code (byte) of the CCW that is built by SAM. The control character sent to the printer is used as follows:

If CTLCHR=YES
SAM uses the control character as the command code. Therefore, the supplied code must be valid for the printer.

If CTLCHR=ASA
SAM translates the control character into the corresponding command code for the printer.

To effect a space or a skip without printing, your program must supply the corresponding control character in the first character position of the output area for the file. The remainder of the area must be all blanks (X'40').

If RECFORM=UNDEF, the length of the record must be at least 2; if RECFORM=VARUNB, it must be at least 6.

For a complete list of all control characters, see z/VSE System Macros Reference.

CNTRL Macro

The macro can be used for forms control on any printer. For output directed to an IBM 4248 printer, the macro can be used to control 4248-specific functions.

The macro normally requires two or three of four possible operands:

- Name of the DTFPR macro

  The name you use when you code the macro to define the printer file. You can specify this name as a symbol or in register notation.
- Mnemonic code for the operation

  One of a set of predetermined codes. For a list of these codes, see the description of the CNTRL macro in https://publibfp.dhe.ibm.com/epubs/pdf/iesmfe81.pdf.
- Operand **n1**

  Normally, a number for stacker selection or for immediate printer carriage control. For an IBM 4248 printer, it may be a code that defines the device operation which is to be performed (EHC for enable horizontal copy, for example).
- Operand **n2**

  This operand applies to delayed spacing or skipping.

If the DTFPR macro for your print file includes the CTLCHR operand, your program can issue a CNTRL for this file only to request an immediate printer operation. Examples of such an operation are:

Space or skip immediate.
Enable or disable horizontal copying on an IBM 4248.

If you use the CNTRL macro to control line spacing or or skipping for a file, your DTFPR for the file must include the CONTROL operand, and it may not include the CTLCHR operand. If control characters are used although CONTROL is specified, SAM treats these characters as data.

Space and Skip Operations

The codes for spacing over a certain number of lines or skipping to a certain line position on the form are SP and SK, respectively. The third operand, n1, controls immediate spacing and skipping (before printing); the fourth operand, n2, controls delayed spacing or skipping (after printing).

The SP and SK operations can be used in any sequence. However:

- Two or more consecutive immediate skips to the same printer result in a single skip-immediate operation.
- Two or more consecutive delayed spaces or skips to the same printer causes SAM to perform only the space or skip operation requested last.

Any other combination of consecutive controls (SP and SK), such as immediate space followed by a delayed skip or immediate space followed by another immediate space, causes both specified operations to be performed.

Printer with a UCS Buffer

You can use the CNTRL UCS=... macro before a PUT for a file to define how data checks are to be handled. With this form of the macro, you can request SAM either to process a data check with an indication given to the operator or to ignore it with a blank printed for the unprintable character. SAM ignores this form of the CNTRL macro if the macro is used for a printer without a UCS buffer (or other than an IBM 3800).

A data check occurs when a character (except X'00 for null or (X'40' for blank) sent to the printer does not match any of the characters in the UCS buffer. On an IBM 3800, a data check occurs when an unprintable character is transferred or when an attempt is made to merge a character with another character different from itself in the same print position.

You can control the handling of data checks by the UCS operand of the DTFPR macro for your file. Specify UCS=ON if you want the console operator to be informed when a data check occurs. Specify UCS=OFF (or nothing) if an unprintable character is to be represented by a blank. If several DTFPRs are assigned to the same physical unit, the UCS specification of the DTFPR opened last determines how a data check is to be handled.

If your program writes to an IBM 3800, the DCHK operand of the SETPRT macro has an effect similar to that of a UCS specification in a DTFPR macro.

You can control the handling of printer data checks (and override your UCS specification in the DTFPR of a file) also as follows:

- By the BLOCK operand of the UCS job control command if your program writes to an IBM 1403U.
- By the NOCHK option of UCB control statement of the SYSBUFLD program.
- By the DCHK operand of the job control statement SETPRT if your program writes to an IBM 3800.

Note: Opening a DTFDI for a UCS printer has the effect of a NOCHK option. This change is effected on the printer and is valid for all DTFPRs assigned to the printer.

FOLD and UNFOLD Codes

Skip this section if your program writes to an IBM 1403, 3203, or 3800.

By way of the CNTRL macro, you can control the printing of lowercase letters. They can be printed as they are or they can be replaced by uppercase equivalents. In your program, issue a CNTRL macro for this purpose before the PUT macro for the lowercase output.

Until your program issues the CNTRL macro, the printing of lower case letters is controlled by the UCB FOLD (or UNFOLD) operand of SYSBUFLD. If FOLD is specified, SAM prints the uppercase equivalent of bits 2 to 7 (for a character other than A through Z, there may not be an uppercase equivalent). If UNFOLD is specified, SAM prints the character equivalent of the EBCDIC byte.

Selective Tape Listing

To use the selective tape listing feature on the IBM 1403, specify the operand STLIST=YES in the DTFPR for your file.

The (optional) operand STLSP of the PUT macro specifies a byte that controls spacing while using the feature. Spacing for up to eight paper tapes may be controlled.

The control byte is set up in virtual storage like any other data byte. You determine the spacing (which occurs after printing) by setting on the bits corresponding to the tapes you want to space. The correspondence between the bits of the control byte and the tapes is as shown below.

Tape Position	Bit No
8 (rightmost)	0
7	1
6	2
5	3
4	4
3	5
2	6
1 (leftmost)	7

Note: Double-width tapes are controlled by the corresponding adjacent bits.

Programming for Output to an IBM 4248

Skip this section, if you use your IBM 4248 like a PRT1 printer.

For your output to make use of the printer's horizontal-copy function, ensure that:

1. The correct FCB image is loaded into the printer's FCB.
2. Horizontal copying is turned on.

You accomplish this by having the required FCB image loaded in response to an LFCB macro. The macro not only causes the requested image to be loaded, it also turns on the horizontal-copy function.

As an alternative to the above method, you can use the following approach:

1. Either:
   - Have your program precede a SYSBUFLD run that loads the correct FCB image into the printer's FCB, or
   - Have your operator load the FCB by issuing an LFCB command.
2. Turn on the horizontal-copy function by a CNTRL macro after the file has been opened. The format of the CNTRL macro for this purpose is:

```
    CNTRL  filename,ORDER,EHC
```

This alternative is of advantage, for example, if you need a copy of only a certain part of your program's printed output. You can turn off the horizontal-copy function for that part of the output of which you do not need a copy. You do this in your program by coding:

```
    CNTRL  filename,ORDER,DHC
```

In the CNTRL macro, you can use register notation to specify the name of your print file, of course.

If your program does not turn off horizontal copying explicitly, the system turns it off automatically when one of the following occurs:

- Your program issues a CLOSE for the printer file.
- End of job step.
- SYSLST is assigned to the printer.

Other IBM 4248-specific functions that you can control by way of CNTRL macros in your program:

- Printing buffered data

  If, at a certain point, your program requires that all data directed to the printer has indeed been printed, code:

  ```
  CNTRL  filename,ORDER,CLRPRT
  ```

- Purging buffered data To purge all data still in the printer's data buffer, code:

  ```
  CNTRL  filename,ORDER,PURDAT
  ```

Error Handling

The ERROPT operand of the DTFPR specifies the action to be taken if an equipment error occurs. You can specify one of the following:

ERROPT=RETRY
Applies only to a file for printing on a PRT1 printer. It specifies that, if an equipment check with command retry is encountered, the command is retried once. SAM cancels the job if the retry is unsuccessful.

ERROPT=IGNORE
Applies only to a card-print file on the IBM 3525. It causes SAM to ignore the error and to place the address of the error record into register 1. The record is thus available for processing. Bit 3 of CCB byte 3 is also set on. You can check this bit and take the suitable action to recover from the error. IGNORE must not be specified for files with two I/O areas or a work area.

ERROPT=name
Applies only to a file for output to a PRT1 printer. If an equipment check with command retry occurs, the command is retried once. SAM cancels the job if the retry is unsuccessful.

For other types of errors, SAM:

1. Issues an error message.
2. Places error information into the CCB part of the DTF table.
3. Gives control to the routine whose name you specified.

In your routine, do not issue any imperative macro instruction for the file that caused the error exit. Your routine can examine the error information (in the two CCB communication bytes) by referring to filename+2. To continue processing at the end of the routine, return to SAM by branching to the address in register 14.

Processing a Console File

The DTFCN macro defines a file that is to be processed for input from or output to the system console. You need not code a separate logic module macro for the file.

To communicate with the operator, your program must use macros with file definitions as follows:

Request Macro
Operand of DTFCN

GET
TYPEFLE=INPUT

PUT
TYPEFLE=OUTPUT

PUTR
TYPEFLE=CMBND

Use the PUTR (PUT with reply) macro to issue a message that requires a response from the operator. The system holds the message on the display screen until the operator has given the required reply. Use PUTR also for messages of fixed-unblocked record format. Issue the macro after your program has built the record.

If you use the PUTR macro, you cannot use register 2 as base register.

In the DTFCN macro, the IOAREA1 operand specifies the name of the I/O area used by the file. The BLKSIZE operand specifies the length of the I/O area. This length may not exceed 256 characters. If your file uses the undefined record format, make sure your longest record fits into the I/O area.

For a PUTR macro, the first part of the I/O area is used for output, and the second part is used for input. The lengths of these parts are specified by the BLKSIZE and INPSIZE operands, respectively.

SAM does not clear the I/O area before or after a message is written to the console. It does not clear the area when a message is canceled and reentered at the console.

Chapter 7. Processing a Device-Independent System File with SAM

Device independence allows you to program as if a certain device were always available. When the program is actually run and the device happens not to be available, the symbolic device name can be assigned easily to some other device. In some cases, the other device may even be of a different type.

The DTFDI macro provides device independence for a file that uses a system logical unit. If several DTFDI macros are assembled within one program and all of them were defined as read only, then just one logic module (DIMOD) is required.

You need not specify DIMOD if the device being used is one of the following:

- A disk device
- A printer of type PRT1, IBM 4248, or IBM 3800,

The support for these types of devices includes pre-assembled logic modules that are automatically loaded during system startup. When a device independent file on a device of these types is opened, the required module is linked to your program. However, consider to include a DIMOD specification in your program if you are not certain which device will be assigned to the logical unit for processing. When the file is opened, the OPEN routines for disk (PRT1 or IBM 3800) override the DIMOD linkage if the logical unit is assigned to such a device.

Use the DTFDI macro to read SYSIPT data if your program might be invoked by a cataloged procedure. In that case, the input data may be part of the procedure.

Restrictions for DTFDI Processing

- Only unblocked records of fixed length are supported.
- For a file on tape:
  - Only forward reading is allowed.
  - No repositioning is done when the file is opened or closed.
- Reading, writing, or checking of standard or user-standard labels for tape or on disk is not supported.
- If ASA control character codes are used in a multitasking environment and more than one DTF uses the same read-only module, overprinting can occur.
- The DLBL statement for a DTFDI file can specify only system files (IJSYSxx); the file name of the DTFDI macro is ignored.
- Combined file processing is not supported for reader-punches.
- Reading of cards is restricted to the first 80 bytes per card.
- The CNTRL and PRTOV macros cannot be used.
- For printers and punches, SAM checks for an ASA control character first and then for an S/370-type control character. Therefore, a valid ASA control character is used as such even if it may also be a valid S/370-type control character. For a list of control characters, refer to z/VSE System Macros Reference.

Record Size

For an **input file** (SYSIPT and SYSRDR), specify the maximum allowable size of 81 bytes and an I/O area, also of 81 bytes. This ensures proper handling of control characters; in this case, the first byte of the I/O area always contains the first byte of data, even if the input consists of 80 data bytes plus one control character.

For an **output file**, each record must include one byte for a control character. The maximum record size for SYSLST is 121 bytes; it is 81 bytes for SYSPCH.

The length of the records is specified by the RECSIZE operand. If you do not define the length of the records, SAM assumes:

```
    80 bytes   for SYSIPT
    80 bytes   for SYSRDR
    81 bytes   for SYSPCH
   121 bytes   for SYSLST
```

Error Handling

The DTFDI operands WLRERR and ERROPT control the processing of record-length and I/O errors.

Wrong-Length Record Errors

The WLRERR operand applies only to input files on devices. It specifies the name of your routine to which SAM passes control when a record of wrong length has occurred.

A wrong-length record error exists, if the length of a record is not the same as that specified in the RECSIZE operand:

- If the record is too short, the first two bytes of the DTF control block contain the residual count, the number of bytes that were not read.
- If the record is too long, SAM sets the residual count to zero, and there is no way to compute the actual size of the record. The number of bytes transferred is equal to the value that you specified in the RECSIZE operand; the remainder of the record is truncated.

When your routine receives control, register 1 contains the address of the record in error. In your routine, you can perform any operation, except one of the following:

- Issue another GET for the file.
- If the file is assigned to a disk, issue any LIOCS macro other than ERET – this would cause your program (or task) to be canceled.

If you specified RDONLY=YES in the DTFDI macro for your file, you must save also the contents of register 13.

At the end of the routine, return control to SAM by a branch to the address in register 14 or by issuing the ERET macro. If you use the ERET macro, you must save the address stored in register 14. When control returns to your program, the next record is available. If you omit the WLRERR operand and a wrong-length record occurs, the action of SAM depends on whether the ERROPT operand is included:

- If the ERROPT operand is included, the record is handled in accordance with your specification in the ERROPT operand.
- If the ERROPT operand is omitted, SAM ignores the wrong-length error and makes the error record available to your program.

Irrecoverable I/O Error

The ERROPT operand applies only to input files. For an output file on most devices, SAM cancels the job after having tried to rewrite the record.

ERROPT specifies the function to be performed for an error block. If an error is detected while reading from tape or disk, SAM attempts to recover from the error. If SAM cannot recover and you did not specify ERROPT, SAM cancels the job.

With the ERROPT operand you can specify the action SAM is to take instead of canceling the job. The three possible specifications are:

ERROPT=IGNORE
Indicates that the error is to be ignored. The address of the error record is made available to your program for processing.

ERROPT=SKIP
Indicates that the error block is not to be made available for processing. SAM reads the next record and processing continues.

ERROPT=name
Indicates that SAM is to pass control to the named routine when an error occurs.

When the routine receives control, register 1 contains the address of the record in error. Do not use the IOREG register to access an error record because the contents of that register may very.

In your routine, you may perform any operation (or simply note the error condition), except any of the following:

- Issue another GET for the file.
- If the file is assigned to a disk, issue any LIOCS macro other than ERET – this would cause your program (or task) to be canceled.

If you specified RDONLY=YES in the DTFDI macro for your file, you must save also the contents of register 13.

At the end of the routine, return control to SAM by a branch to the address in register 14 or by issuing the ERET macro. If you use the ERET macro, you must save the address stored in register 14. When control returns to your program, the next record is available.

End-of-File Handling

The EOFADDR operand specifies the name of your end-of-file routine. The routine is required if SYSIPT or SYSRDR is specified as device address.

SAM passes control to this routine when end of file occurs. In this routine, you can perform any operations necessary for the end-of-file condition (you generally issue the CLOSE macro). An end-of-file condition exists when the following occurs for SYSIPT or SYSRDR on a device as indicated:

- A card reader: a /* in positions 1 and 2 of the record.
- A tape: a /* in positions 1 and 2 of the record or a tapemark.
- A disk: a /* in positions 1 and 2 of the record or an end-of-file record.

Chapter 8. Requesting Control Functions

This topic discusses how to use the control-function macros by major functions such as the loading of a program, control of virtual storage, and multitasking.

Program Loading

A program (or the first phase of a program) is normally loaded by the system in response to the job control EXEC statement. However, by way of a FETCH, LOAD, or CDLOAD macro, a phase in control can request another phase to be loaded. The phase to be loaded may be stored either:

- In a sublibrary that is accessible from the partition in which the requesting phase resides, or
- In the shared virtual area (SVA).

For information on 31-bit addressing, see z/VSE Extended Addressability.

The Load Request

The FETCH Macro

The macro gives control to the phase just loaded.

The load point and the entry point of the requested phase are the addresses determined when the phase was link-edited. You may specify a different entry point in the FETCH macro. However, this entry point must be in the same partition as the requesting phase.

The macro cannot request a self-relocating phase to be loaded.

The LOAD Macro

Control remains with the phase that issued the macro.

The load and entry points of the requested phase are the addresses determined when that phase was link-edited. After having loaded the phase, the system loads the address of the entry point of that phase into register 1. Your program must decide at which point the newly loaded phase is to receive control.

The macro allows you to override the link-edited load point of the requested phase. If you do this, the system relocates the address of the entry point. For a relocatable phase, the system relocates also all addresses to reflect the current load point of the phase. For a non-relocatable or self-relocatable phase, the system relocates only the entry point address.

The CDLOAD Macro

Control remains with the phase that issued the macro.

The macro requests a phase to be loaded into the GETVIS area of the partition in which your program runs. For more information about the partition GETVIS area, see z/VSE Guide to System Functions.

The macro causes the system to:

1. Allocate the required amount of virtual storage in the partition GETVIS area.
2. Load the phase.
3. Load the address of the entry point of the phase into register 1.

Your program must decide at which point the newly loaded phase is to receive control. The entry point of the phase is at the same relative distance from the actual load point as it was from the link-edited load point.

The CDDELETE Macro

The macro deletes a phase previously loaded by the CDLOAD macro into the partition GETVIS area. Deletion means that the phase load count is decreased by one. If the load count is zero, the GETVIS storage occupied by the phase will be freed.

Load Request for a Phase in the SVA

A requested phase may reside in the shared virtual area (SVA). In this case, no actual load operation takes place.

For a **FETCH macro**, the system transfers control to the entry point of the requested phase in the SVA and processing continues.

For the **LOAD or CDLOAD macro**, the entry-point address of the requested phase is returned in register 1. If the CDLOAD macro is issued by a program running in real mode, the system loads a copy of the phase into the partition GETVIS area.

Fast Loading of Frequently Used Phases

The system directory list (SDL), located in the SVA, contains directory entries for frequently used phases. Operands of the FETCH and LOAD macros cause the SDL to be searched before a search of any private sublibrary directory is performed. This helps reduce the time needed for locating the requested phase.

For a phase which is being loaded frequently during execution of only one program, consider using the GENL macro rather than including the directory entry of the phase in the SDL. The GENL macro generates a "local" directory list within the partition. On the first LOAD or FETCH for the phase, the system supplies its entry with information that helps reduce access time on any subsequent LOAD or FETCH for the phase.

Figure 39 on page 100 shows how to use the LOAD macro together with a local directory list. The first LOAD macro locates a certain entry (PHASEX) of the directory list but does not cause the phase to be loaded into storage (TXT=NO is specified).

```
         ...
         LOAD  PHASEX,LIST=GENLIST,TXT=NO
         LR    2,0             GET PTR TO DIRECTORY ENTRY
         TM    16(2),X'06'     PHASE FOUND?
         BO    NOTFOUND        NO
         TM    16(2),X'12'     PHASE IN SVA?
         BO    NOLOAD          YES, BRANCH AROUND LOAD
         LA    0,LOADPT
         LOAD  (2),(0),DE=YES
NOLOAD   EQU   *               RFG.1 POINTS TO ENTRY POINT
         ...
GENLIST  GENL  PHASEX,...
         ...
LOADPT   DS    0D              LOAD POINT OF OVERLAY PHASE
```

Figure 39. Example for Using the LOAD Macro with a Local Directory List

Virtual Storage Control

The macros that are discussed in this section perform the following services:

- Fix a page in processor storage and later free that page for normal paging.
- Determine the mode of execution of a program.
- Request partition-related information, such as partition boundaries.
- Reduce the number of page faults.
- Allocate and release virtual storage dynamically.

This section assumes that you are familiar with the virtual storage concept as implemented for VSE and described in z/VSE Guide to System Functions. For information on 31-bit addressing, see z/VSE Extended Addressability.

Fixing and Freeing a Page in Processor Storage

VSE ensures that instructions and related data are in processor storage (sometimes also called real storage) when they are being used. However, data areas could be paged out during an I/O operation if nothing were done to keep them in processor storage during the entire operation. The system fixes I/O areas in processor storage for the duration of the I/O operation.

A program may have sections which cannot tolerate paging, and these sections are not necessarily kept in processor storage by the system. For instance, programs that control time-dependent I/O operations cannot tolerate paging. To avoid a page fault in such a program, you must fix the affected pages in processor storage. You do this in your program by issuing the PFIX macro.

You can use the **PFIX macro** if a certain number of page frames are reserved for fixing pages in your program's partition. Normally, this is taken care of during system start-up by a SETPFIX or ALLOC R job control statement.

The macro fixes one or more pages in processor storage, and these pages need not be contained in contiguous page frames. The system keeps a count of the number of times a PFIX was issued for a certain page without this page having been freed. A page that is fixed more than once without having been freed (via the PFREE macro) is not brought in repeatedly and given additional page frames. Instead, the system increments the page-fix counter for that page by 1, and the page remains in the same page frame. The counter must not exceed 32,767.

The location of the real storage to be fixed can be indicated by the RLOC operand: RLOC=BELOW indicates that the virtual address range has to be fixed in real storage frames which are below the 16MB boundary, RLOC=ANY indicates that it can be fixed in any frame.

The **PFREE macro** does not necessarily free a page for paging out. Each time a PFREE is issued, the system decrements the count of fix requests by 1. Not until the counter for a page reaches zero can the page be paged out. At the end of a job step, all pages that have been fixed during the job step are freed by the system.

Use the PFREE macro as soon as possible after a PFIX to make the originally fixed page frames available to all programs running in virtual mode.

When your program receives control again, register 15 contains a return code. This code indicates successful or unsuccessful completion of the request. For more information about these codes, see z/VSE System Macros Reference.

Figure 40 on page 102 is an example for using the PFIX and PFREE macros. It shows how your program can use the return code to set up a branch to parts of the program that handle the various conditions.

```
          ...
FIXER     PFIX   ARTN,ARTNEND+2 FIX ARTN IN STORAGE
          B      *+4(15) BRANCH ACCORDING TO RETURN CODE
          B      HERE    CONTINUE IF OK
          B      NOPAGES BRANCH IF PARTITION TOO SMALL
          B      WAIT    BRANCH TO WAIT UNTIL PAGES ARE
*                        FREE
          B      CANCL   BRANCH IF PFIX ADDRESSES ARE
*                        INVALID
HERE      BAL    14,ARTN
          PFREE  ARTN,ARTNEND+2
*                        FREE ROUTINE - GETS CONTROL
*                        AFTER EXECUTION
ARTN       ...      Time dependent processing
           ...      which cannot tolerate its
           ...      code to be paged out during
           ...      program execution.
ARTNEND   BR     R14     RETURN
NOPAGES   LA     R1,OPCCB
          EXCP   (1)     WRITE MESSAGE TO OPERATOR
          WAIT   (1)     WAIT FOR COMPLETION
CANCL     CANCEL ALL
WAIT        (routine to free other pages)
END       EOJ
OPCCB     CCB    SYSLOG,OPCCW
OPCCW     CCW    X'09',MSG,X'20',61
MSG       DC     CL32'AM CANCELING. ENLARGE REAL'
          DC     CL29'STORAGE AND RESTART THE JOB'
           ...
```

Figure 40. PFIX and PFREE Example

Determining the Run Mode of a Program

Your program may do different processing for virtual and for real mode. In that program, you can issue the RUNMODE macro to find out which mode of operation is being used. The system returns this information in register 1.

Extracting Partition-Related Information

You may have a program for which you want to manage the storage available to the partition in which the program runs. A typical example is a program that does a lot of I/O. The program should have as many buffers available as the size of the partition permits. To do such storage management, the program needs to know what the partition's boundaries are. Use the EXTRACT ID=BDY macro to retrieve this information.

To interpret the retrieved information, use the DSECT generated by the MAPEXTR ID=BDY (or MAPBDY/MAPBDYVR) macro.

Reducing the Number of Page Faults

The macros discussed here are provided primarily for optimizing performance beyond the level of optimization by the system. The support allows you to control the amount of paging in your system.

Releasing a Page

The RELPAG macro informs the system that your program no longer needs the contents of one or more pages, and that these contents need not be saved on the page data set. Page frames occupied by these pages can be claimed for use by other pages immediately. This can reduce page-I/O activity.

Forcing a Page-Out

The FCEPGOUT macro informs the system that one or more pages are not needed until a later stage of processing. The pages are given the highest page-out priority. As a result, other pages which are still needed for immediate reference remain in processor storage.

All program pages paged out because of a FCEPGOUT request are saved by writing them onto the page data set (as opposed to a RELPAG request).

Requesting a Page-In

The PAGEIN macro allows you to request that one or more program pages are paged-in in advance to avoid page faults when the requested pages are needed. Unlike the PFIX macro, the PAGEIN macro does not fix the requested pages. Therefore, issuing the macro does not ensure that the paged-in pages are in processor storage when they are needed. If the requested pages are already in processor storage when the macro is issued, they are given the lowest priority for page-out.

Allocating Virtual Storage Dynamically

With the GETVIS and FREEVIS macros, a program can dynamically acquire and release blocks of virtual storage. The system allocates such storage either from the GETVIS area of your partition or of the SVA, or from the dynamic space GETVIS area.

You can access the partition GETVIS area **above** 16MB or the 31-bit SVA by using the LOC operand of the GETVIS macro. For details refer to z/VSE Extended Addressability.

A minimum GETVIS area is always reserved in a partition as long as a job in that partition runs in virtual mode. This minimum can be enlarged by the SIZE command, for example. For more information about defining a GETVIS area, see z/VSE Guide to System Functions.

Program Communication

This section discusses the support available for communication between job steps or jobs within one partition. You can do this in either or both of the following ways:

- Communication via the partition communication region

  A convenient way if no more than 11 bytes of information are to be communicated from one step of a job to another.
- Communication via a communication area

  A convenient way if information up to a length of 256 bytes is to be communicated.

The section includes examples for the use of the available communication macros.

Communication via the Partition Communication Region

The system maintains this area for each of its batch partitions. The macros COMRG and MVCOM are available to access this area:

The COMREG macro
The macro places the address of the partition communication region into register 1. By using that register as base register, your program can read any of the fields listed and discussed below.

The MVCOM macro
The macro modifies the contents of field COMUSCR (bytes 12 through 22) and UPSI (byte 23) of the partition communication region.

The user-oriented fields of the communication region are (offsets in hexadecimal and field lengths in decimal notation):

Field Name	Offset	Length (Bytes)	Information
JOBDATE	00	8	Calendar date. Supplied by the system whenever a JOB statement is encountered. The format of the date is either mm/dd/yy or dd/mm/yy, where: dd = day mm = month yy = year This date can be temporarily overridden by a DATE statement.
	08	4	Reserved.
COMUSCR	0C	11	User area. Available for communication within a job step or between job steps. All bytes of the field are set to zero whenever a JOB or end-of-job statement for a job is encountered.
UPSI	17	1	UPSI (user program switch indicators). Set to binary zero when a JOB or end-of-job statement for the job is encountered. Initialized by the UPSI job control statement.
COMNAME	18	8	Job name as found in the JOB statement for the job.
PPEND	20	4	Address of the partition's uppermost byte available to your problem program.
HIPHAS	24	4	Address of the uppermost byte of the phase loaded into the partition by the last FETCH or LOAD request in the job.
HIPROG	28	4	Highest ending virtual storage address of the longest phase, starting with the same four characters as the root phase (operand on the EXEC statement) and residing in the same sublibrary as the root phase. If the root phase is in the SVA, the partition start address plus 2K is used.
LABLEN	2C	2	Length of program label area.
IJBHPHLA	244	4	Highest end address of any phase loaded up to now in this job step.

Communication via a Communication Area

The JOBCOM macro provides for communication between job steps or jobs of a partition. For **dynamic** partitions, this applies only to VSE jobs and job steps within one VSE/POWER job, because a dynamic partition and its control blocks are only existent during the execution of a VSE/POWER job.

Information to be communicated is stored in a 256-byte area. The system provides such an area for each partition.

The JOBCOM macro either moves information to that area or retrieves information previously stored there by another program. The area remains unchanged from one job (or job step) to the next. Unless it is

modified (by way of a JOBCOM macro), the content of the area remains unchanged over any number of jobs.

Before your program issues the JOBCOM macro, it must:

1. Provide an 18-word register save area. For more information about this area, see “Register Save Area” on page 114.
2. Load the address of this area into register 13.

Program-Communication Examples

The following example shows how to move three bytes from the symbolic location DATA into bytes 16 through 18 of the communication region:

```
    MVCOM 16,3,DATA
```

The following example shows how eight bytes of information are stored in the first eight bytes of the system-supplied area. The remaining 248 bytes of that area remain unchanged.

```
          ...
          LA     13,JCOMSAVE
          JOBCOM FUNCT=PUTCOM,                          X
                 AREA=JCOMINFO,LENGTH=8
          ...
JCOMSAVE  DS     18F
JCOMINFO  DC     C'ABCDEFGH'
          ...
```

Assigning and Releasing an I/O Unit

Dynamic assignment and release of an I/O unit can be useful in long-running and complex applications that require a unit only for a short time (to store intermediate processing results, for instance). The section “Logical Units” on page 36 briefly discusses the concept of logical unit assignment.

Your program can use the ASSIGN macro to dynamically assign an I/O device not currently owned by one of your system's partitions. It can use the macro also to dynamically release the unit when your program has no further use for it.

If an assignment made with the ASSIGN macro is not released (again with the ASSIGN macro), this assignment is reset along with all other temporary assignments when the next EOJ statement (/&) is encountered. If you use the ASSIGN macro to release a tape drive, ensure that the logical unit number of the unit to be released is still stored in the parameter list used by the macro. Your program can interpret the list with the help of the mapping DSECT generated by the ASPL macro.

Figure 41 on page 106 shows the layout of the parameter list. The list is generated by the ASPL macro if you specify DSECT=YES. If you specify DSECT=NO (the default), the parameter list starts with the following statements:

```
name      DS      0XL5
ASGFUNCT  DC      XL1'00'
```

```
Field Name              Explanation

ASGLIST  DS    0XL5     ASSIGN parameter list
ASGFUNCT DS    XL1      ASSIGN function code, see below
ASGDPT   EQU   X'80'    Temporary programmer logical unit-specific disk
ASGDST   EQU   X'84'    Temporary system logical unit-specific disk
ASGDSP   EQU   X'8C'    Permanent system logical unit-specific disk
ASGTPT   EQU   X'40'    Temporary programmer logical unit-tape
                        (no mode defined)
ASGTPTM  EQU   X'60'    ASGTPT with mode specification
*
ASGTPD   EQU   X'42'    Temporary programmer logical unit-specific
                        tape (no mode defined)
ASGTPDM  EQU   X'62'    ASGTPD with mode specification
*
ASGTPS   EQU   X'43'    Temporary programmer logical unit-specific
                        tape (no mode). Log. unit must be unassigned.
ASGTPSM  EQU   X'63'    ASGTPS with mode specification
*
ASGUAP   EQU   X'28'    Unassign programmer logical unit
ASGUAS   EQU   X'2C'    Unassign system logical unit
ASGCHG   EQU   X'10'    Change temporary to permanent assignment
ASGURT   EQU   X'02'    Specific device other than temporary disk/tape
*                       programmer logical unit
ASGUEX   EQU   X'03'    ASGURT plus additional information
ASGLUNO  DS    0XL2     Logical unit number
ASGCLASS DS    XL1      Logical unit class
ASGPROG  EQU   X'01'    Programmer class
ASGSYST  EQU   X'00'    System class
ASGLUNDX DS    XL1      Logical unit index
ASGCUU   DS    XL2      Physical unit number
ASGLNG   EQU   *-ASGFUNCT  Length of ASPL without additional information
ASGCODE  DS    XL1      Additional information
ASGEXCLU EQU   X'01'    Exclusive I/O assignment
ASGMODE  DS    XL1      Mode for tape units
ASGLNGE  EQU   *-ASGFUNCT  Length of ASPL with additional information
*
```

Figure 41. Parameter List Generated by the ASPL Macro

Explanation of Function Codes in Detail

ASGDPT
Temporary assignment of a disk unit to a programmer logical unit. ASGCUU must contain a valid disk cuu number. ASGLUNO will contain a programmer logical unit number after successful execution.

ASGDST
Temporary assignment of a disk unit to a system logical unit (normally used for system functions). ASGCUU must contain a valid disk cuu number. ASGLUNO will contain a system logical unit number after successful execution.

ASGDSP
Permanent assignment of a disk unit to a system logical unit (normally used for system functions). Note that the assignment will not be released at end-of-job. ASGCUU must contain a valid disk cuu number. ASGLUNO will contain a system logical unit number after successful execution.

ASGTPT
Temporary assignment of any tape unit to any programmer logical unit. ASGCUU will contain a tape cuu number and ASGLUNO will contain a system logical unit number after successful execution.

ASGTPTM
This is the same function as ASGTPT, except that a mode character is to be specified in ASGMODE. This is the same mode that can be given with the ASSGN job control statement. An assignment is only done if a free tape unit with that mode is found. During assignment the mode is set.

ASGTPD
Temporary assignment of a tape unit to a programmer logical unit. ASGCUU must contain a valid tape cuu number. ASGLUNO will contain a programmer logical unit number after successful execution.

ASGTPDM
This is the same function as ASGTPD except that a mode character is to be specified in ASGMODE. This is the same mode that can be given with the ASSGN job control statement. An assignment is only done if the specified tape unit can handle the mode. During assignment the mode is set.

ASGTPS
Temporary assignment of a tape unit to a specified programmer logical unit. ASGCUU must contain a valid tape cuu number. ASGLUNO must contain a Dprogrammer logical unit number.

ASGTPSM
This is the same function as ASGTPS except that a mode character is to be specified in ASGMODE. This is the same mode as the one that can be given with the ASSGN job control statement. An assignment is only done if the specified tape unit can handle the mode. During assignment the mode is set.

ASGUAP
Unassign a programmer logical unit. ASGLUNO must contain a programmer logical unit number that is assigned to a device.

ASGUAS
Unassign a system logical unit. ASGLUNO must contain a system logical unit number that is assigned to a device.

ASGCHG
Change temporary assignment to permanent. (Note that the assignment is not reset at end-of-job.) ASGLUNO must contain a logical unit number that is temporarily assigned to a device.

ASGURT
Temporary assignment to a programmer logical unit of a device that is neither disk nor tape. ASGCUU must contain a valid cuu number (not disk or tape). ASGLUNO will contain a programmer logical unit number after successful execution.

ASGUEX
This is the same function as ASGURT, except that additional information must be specified in ASGCODE. The only information so far is ASGEXCLU which specifies that the assignment will be done exclusively and will be rejected if this is not possible.

Figure 42 on page 108 is a skeleton coding example. It shows how a tape drive can be assigned and released dynamically.

```
ASSIGN      ASPL    DSECT=YES              GENERATE MAPPING DSECT FOR
RX          EQU     10                     PARAMETER LIST
                     ...
            XC      ASPLX,ASPLX            CLEAR PARAMETER LIST
            LA      RX,ASPLX               ESTABLISH ADDRESSING
*                                          AND MAPPING TO THE
            USING   ASSIGN,RX              PARAMETER LIST
            MVI     ASGFUNCT,ASGTPT        INDICATE ASSIGN:
*                                          TAPE PROG. LOGICAL UNIT - TEMP.
            ASSIGN  ASPL=(RX),SAVE=SAVEAREA
*                                          TEMPORARILY ASSIGN ANY AVAIL.
*                                          PROGRAMMER LOGICAL UNIT TO ANY
*                                          AVAILABLE TAPE DRIVE
            MVC     MT06IN+7(1),ASGLUNDX   PUT LOGICAL UNIT INTO TAPE
*                                          DTFMT (MT06IN)
            OPEN    MT06IN                 PERFORM DESIRED I/O
GET00       GET     MT06IN,INAREA          FUNCTIONS
            B       GET00
ENTAPE      CLOSE   MT06IN
            LA      RX,ASPLX               ENSURE ADDRESSABILITY
            MVI     ASGFUNCT,ASGUAP        INDICATE UNASSIGN
            ASSIGN  ASPL=(RX),SAVE=SAVEAREA
*                                          FREE TAPE AND LOGICAL UNIT
*                                          (ASGLUNO STILL INTACT)
            EOJ
ASPLX       DS      CL(ASGLNG)             DEFINE PARAMETER LIST
*                                          IN LENGTH OF ASPL
SAVEAREA    DS      18F                    DEFINE SAVE AREA
MT06IN      DTFMT   BLKSIZE=800,                                       X
                    DEVADDR=SYS001,                                    X
                    EOFADDR=ENDTAPE,                                   X
                    FILABL=STD,                                        X
                    IOAREA1=INAREA,                                    X
                    RECFORM=FIXBLK,                                    X
                    RECSIZE=80,                                        X
                    WORKA=YES,                                         X
                    TYPEFLE=INPUT,                                     X
                    REWIND=UNLOAD
INAREA      DC      CL80' '
```

Figure 42. Example for a Device Assignment

A programmer logical unit can also be released from within a program by the RELEASE macro. You can use the macro for a unit that is assigned to the partition in which your program runs.

The macro unassigns the specified programmer logical unit(s), unless they are assigned permanently. If you use the macro, your program should inform the system operator by a message that the assignment was released.

Timer Services

VSE provides the following timing facilities:

- Time-of-day clock
- Interval timer

Time-of-Day Clock

The time-of-day (TOD) clock is a standard hardware feature. By issuing the GETIME macro, your program can obtain the time of the day. The system returns the requested time in accordance with your specification in the macro in one of the following formats:

- As a packed decimal number in the form hhmmss, where:

hh	=	hours
mm	=	minutes

ss	=	seconds

- As a binary number in seconds.
- As a binary number in 1/300 seconds.
- As a binary number in microseconds.

Interval Timer

This support is independent of the time-of-day clock; the use of the interval timer and of GETIME have no effect on one another.

Your program (or one of its tasks) can set a real time interval, in seconds, or 1/300 of a second, by using the SETIME macro. The maximum valid interval is either of the following:

55924 (equivalent to 15 hours, 32 minutes, 4 seconds), if expressed in full seconds.
8388607 (equivalent to 7 hours, 46 minutes, 2 seconds, approximately), if expressed in 1/300 of a second.

Expiration of the specified interval causes an external interrupt.

When the interrupt occurs and your program has set up linkage to a timer exit routine via an STXIT IT macro, your program's timer exit routine receives control. At the end of the timer exit routine (statement EXIT IT), control is returned to the point of interruption.

Waiting for a Time Interval to Elapse

When processing depends on the expiration of a time interval, your program can issue a WAIT macro to suspend processing until this interval has elapsed.

The SETIME macro, which you use to define the interval, passes to the system the name of the timer event control block (defined by a TECB macro) to be posted when the interval has elapsed. The WAIT macro specifies the same TECB and passes control to your system's supervisor. This allows another task in your system to gain control over the CPU. When the interrupt occurs, the event bit in the TECB is turned on, and the task that issued the SETIME and WAIT macros is made ready to proceed.

Following is a skeleton sample program that waits for a time interval to expire:

```
         ...
         START 0
         ...
STIMER   SETIME 30,TECB1  START 30 SECOND INTERVAL
         ...
         ...   Normal processing, not time-dependent
         ...
         WAIT TECB1       WAIT FOR TIMER END
         ...
         ...   Time-dependent processing
         ...
TECB1    TECB
         ...
         END
```

Getting the Unexpired Time

The task that issued the SETIME macro can find out how much of the interval is yet unexpired. The TTIMER macro is available for this purpose. The macro returns the unexpired time in hundredths of seconds in register 0 without disturbing the interval timer function.

If the TTIMER macro includes the operand CANCEL, a previously issued SETIME macro is canceled and the remaining time is not returned.

Linkage to User Exit Routines

You use the STXIT macro to set up linkage to a user-written exit routine.

The macro specifies the condition, a two-character code, for which control is to be transferred to the related exit routine. These conditions are:

IT
Interval timer external interrupt

AB
Abnormal end of the program

PC
Program-check interrupt

OC
Operator communication interrupt

The related exit routines are discussed separately in subsequent sections.

Before passing control to a user-exit routine, the system sets up registers 0, 1, and 15. For the contents of these registers, see the description of the STXIT macro in z/VSE System Macros Reference.

At different points in time, different processing routines such as various levels of user routines, or data management routines, might be active. Therefore, the contents of registers 2 to 14 are unpredictable, and addressability must be established within the exit routine.

To return from a user exit routine, use the EXIT macro.

Interval-Timer User Exit

In your program, you may want to do certain processing when a certain amount of time has elapsed. For this purpose, you use the STXIT IT macro to set up linkage to an interval-timer exit routine. When this routine has completed processing, an EXIT IT macro returns control to your program at the next instruction that is to be executed.

Note: If your program uses VTAM, the exit routine does not receive control as long as VTAM is processing any request on behalf of your program. The exit routine will receive control when VTAM has completed the request.

Figure 43 on page 111 shows how to code the STXIT IT macro for a user-exit routine to receive control every half hour during processing.

```
TIMECHK   START X'78'
          BALR  R9,0
BASEADDR  EQU   *
          USING BASEADDR,R9       ESTABLISH ADDRESSABILITY
          STXIT IT,TIMINTR,TIMSA  SET UP LINK TO TIMER
*                                 ROUTINE
          SETIME 1800             TIMER INTERRUPT EVERY 30 MINUTES
           ...
PROCESS   EQU   *
           ...
(perform normal processing)
           ...
          B     PROCESS
*
* TIMER INTERRUPT ROUTINE
*
TIMINTR   EQU    *
          BALR  R9,0
          L     R9,ABASE-*(R9)    ESTABLISH ADDRESSABILITY
           ...
(perform IT exit processing)
           ...
          SETIME 1800             SET UP NEXT INTERVAL
          EXIT   IT               RETURN TO POINT OF INTERRUPTION
*
*
* CONSTANTS
*
ABASE     DC      A(BASEADDR)
TIMSA     DC      9D'0'           IT-EXIT ROUTINE SAVE AREA
          MAPSAVAR                MAPPING OF EXIT SAVE AREA
R9        EQU     9
          END
```

Figure 43. Example of Using the Interval Timer Exit

Following are some multitasking considerations. The main task or any subtask in a partition or both may issue a SETIME macro. Each may also issue an STXIT macro to set up linkage to a common user-exit routine, if this routine is reenterable and each task has its unique register-save area. For a discussion of this area, see “Register Save Area” on page 114.

Figure 44 on page 111 illustrates this approach.

```
MAINTASK START 0
           ...
         STXIT IT,STRTER,MTSKSA
         SETIME 300              SETS MAIN TASK TIMER
*                                TO 5 MINUTES
         ATTACH SUBTASK1,SAVE=SAV1
         ATTACH SUBTASK2,SAVE=SAV2
          ...
* IT USER EXIT ROUTINE
STRTER    ...         Reenterable routine
          ...         Set up addressability
          ...
         EXIT IT
SUBTASK1 STXIT IT,STRTER,STSK1SA USE SAME EXIT ROUTINE
         SETIME 400              SET TIME INTERVAL
          ...
         DETACH
SUBTASK2 STXIT IT,STRTER,STSK2SA USE SAME EXIT ROUTINE
         SETIME 500              SET TIME INTERVAL
          ...
         TTIMER CANCEL           CANCEL INTERVAL THIS
*                                TASK ONLY
          ...
         DETACH
MTSKSA   DS   9D
STSK1SA  DS   9D
STSK2SA  DS   9D
SAV1     DS  16D
SAV2     DS  16D
         MAPSAVAR                MAPPING OF EXIT SAVE AREA
```

Figure 44. Example of Multi-Task Linkage to a Common Exit Routine

Abnormal-End User Exit

The STXIT AB macro sets up or removes linkage to a user-written routine that gets control if the issuing program should end abnormally for a reason other than a self-requested termination. In this routine, you can do any necessary housekeeping such as closing files and writing messages before the program ends.

However, in the exit routine, do not use any of the macros STXIT, SETIME, and SETT. Also, do not use the macros LOCK or ENQ for a resource that is held by the main task; this might result in a deadlock.

After your abnormal-end routine has performed the necessary action, the routine may do either of the following:

- End the task with one of these macros: CANCEL, DETACH, DUMP, JDUMP, EOJ. The macros DUMP and JDUMP are discussed under "Requesting Storage Dumps" on page 132.
- Resume processing by way of the EXIT AB macro if the routine is owned by your program's main task.

After EXIT AB, the routine continues with the next instruction.

For a main task, the **whole job** is canceled if OPTION=DUMP has been specified explicitly or by default. Only the **current job step** is canceled if OPTION=NODUMP is effective and the termination macro used was either DUMP or EOJ.

If OPTION=EARLY is specified in the STXIT AB macro, your exit routine will be invoked for any type of cancellation (normal or abnormal) and, for a main task, before its subtasks are canceled.

Program-Check User Exit

The STXIT PC macro sets up linkage for an exit routine to receive control when a program check other than a page fault occurs. The routine can analyze the interrupt-status information and the contents of the general registers stored in the user's save area of the routine.

If an error condition caused the interrupt, your exit routine can correct the error or decide to ignore it, depending on the severity of the error. Your routine can return control to the interrupted program or request your program to be canceled. After the failing instruction, EXIT PC returns control to the interrupted program. Figure 45 on page 112 shows an exit routine that recovers from a program check caused by an attempt to divide by zero. In this example, any other errors causing a program check result in the user save area being dumped before the job ends.

```
DIVTEST  CSECT
          ...         Set up addressability
          ...
         STXIT PC,PCRTN,PCSAV    SET UP PROGRAM CHECK LINK
          ...
         LM    R2,R3,DIVIDEND    LOAD FOR DIVIDING
         D     R2,DIVISOR        DIVIDE
          ...
*        USER'S PROGRAM CHECK ROUTINE
PCRTN     ...         Set up addressability
          ...
         SR    R5,R5             CLEAR REGISTER 5
         CL    R5,DIVISOR        CHECK FOR ZERO DIVISOR
         BNE   CANCELR           IF NOT CLEAR FILES & CNCL
          ...
          ...         Special recovery routine
          ...
         EXIT   PC               RETURN TO NORMAL PROC
CANCELR  PDUMP  PCSAV,PCSAV+71   DUMP SAVE AREA
          ...
          ...         Close files and do other
          ...         housekeeping
          ...
          ...         Equates and storage definitions
          ...
         CANCEL ALL
```

Figure 45. Example of an Exit Routine Processing a Program Check

Operator-Communication User Exit

You can provide for direct communication between the operator and your program by issuing an STXIT OC macro. You can use the macro only in the main task of your program.

To initiate communication, the operator enters MSG followed by the partition identifier (such as BG or F2). This sets up the linkage to your program's operator-communication (OC) exit routine, which may perform any processing.

In the MSG command the operator can specify data to be passed to the OC exit routine. When you define the OC exit in the STXIT macro, you must code the MSGDATA operand to indicate that the exit is prepared to retrieve data from the MSG command. You must code the MSGPARM operand to pass routing and correlation parameters associated with the MSG command to the OC exit in the user save area.

An operator communication exit routine is performed asynchronously with the main routine of your program. Therefore, be careful when using the same resources (such as data and instructions) in both routines. To be on the safe side, use the data management routines (DTFCN), as described under “Processing a Console File” on page 93.

EXIT OC returns control to the interrupted program.

Note: If your program uses VTAM, your exit routine does not get control as long as VTAM is processing a request for your program. Your exit routine receives control when VTAM has completed the program's request.

Ending a Job Step

Normal End of the Main Task

The normal way of ending the main (or only) task of your program is to issue the EOJ macro. An EOJ macro in the main task of the last step of a job ends the job.

Through the EOJ macro, your program informs the system that processing for the current job step is finished. At this time, subtasks should no longer be attached. If nevertheless one is, the system considers the EOJ from the main task as an abnormal-end condition for the subtask. If the subtask provides an STXIT AB routine, that routine receives control.

Program-Requested Abnormal End

To end a task abnormally, you may use one of the macros discussed under “Requesting Storage Dumps” on page 132. A CANCEL macro in the main task cancels all tasks currently active in the partition.

Program Linkage

A program may consist of several phases or routines produced by language translators and combined by the linkage editor.

The CALL, SAVE, and RETURN macros are used for linkage between routines in storage and within the same or different phases. These macros set up linkage with conventional register and save area usage. They allow control to be passed from one routine to another or from one phase to another. They also allow parameters to be passed.

Linkage can be set up through as many levels as necessary, and each routine may be called from any level. The routine given control during a job step is initially a **called** program. During execution of that program, the services of another routine may be required, at which time the current program becomes a **calling** program.

Linkage Registers

To standardize branching and linking, certain registers are assigned specific roles:

Register 0 – Parameter register
Contains a parameter value to be passed to the called program.

Register 1 – Parameter (-list) register
Contains either of the following:

- A parameter value to be passed to the called program.
- The address of a parameter list to be passed to the called program.

Register 13 – Save area address register
Contains the address of the register save area to be used by the called program.

Register 14 – Return register
Contains the address of the location in the calling program to which control is to be returned when processing by the called program is finished.

Register 15 – Entry-point register
Contains the address of the entry point into the called program.

Registers 0, 1, and 13 through 15 are known as the linkage registers. Before a branch to another routine, the calling program is responsible for the following:

1. Loading the address of a register save area into register 13.
2. Loading, into register 14, the address to which the called program is to return control.
3. Loading, into register 15, the address of the called program's entry point.
4. Loading parameters into register 0 or register 1 or both, or loading the address of a parameter list into register 1.

Register Save Area

A called program should save and restore the contents of the linkage registers as well as any register that it uses. Certain conventions exist for the saving of register:

1. The calling program provides a save area and places the address of this area into register 13 before issuing the call.
2. When having received control, the called program stores (saves) the registers in the save area provided by the calling program.

A save area occupies nine doublewords and is aligned on a doubleword boundary. For programs to save registers in a uniform manner, the save area has a standard format as shown and described below.

Word Displ. (hex)
Contents

1 0
An indicator byte followed by three bytes that contain the length of allocated storage. Use of these fields is optional, except in programs written in PL/I.

2 4
The address of the save area provided by the higher-level calling program (or routine). The address is passed to the called program in register 13. The calling program must store the address into this field before it loads the address into register 13.

Strict adherence to this convention allows this field of the save area to be used as a program-call trace.

3 8
The address of the save area of the next lower level program (or routine), unless this called program is at the lowest level and does not have a save area. The called program requires a save area only if it is also a calling program; thus, the called program, if it contains a save area, stores the address of that area in this word.

4 0C
The contents of register 14 – The address to which the called program (routine) returns control. The called program may save the return address in this word.

5 10
The contents of register 15 – The address of the entry point of the called program. Register 15 contains this address when control is given to the called program. The called program may store the entry-point address in this word.

6-18 14
The contents of registers 0 through 12 – The called program should store the contents of these registers: register 0 into word 6, register 1 into word 7, and so on. The program must store the contents of those registers which it modifies. However, the called program should not save and restore a register that it uses to pass the result of certain processing (see also the coding example in Figure 46 on page 116).

Linkage Macros

The macros CALL, SAVE, and RETURN help you code direct linkage. Before you issue the CALL macro, you need to code only one other instruction: an instruction to load the address of the calling program's save area into register 13.

Figure 46 on page 116 is an example for the use of the three macros.

The CALL Macro

The CALL macro correctly loads registers 14 and 15 and, if parameters are passed, register 1. It then passes control to a specified entry point in the called program.

In the following example, EX1 gives control to an entry point named ENT:

```
EX1 CALL ENT
```

In the example below, EX2 gives control to an entry point whose address is contained in register 15:

```
EX2 CALL (15),(ABC,DEF)
```

Two parameters, ABC and DEF, can be accessed by the called program. When the macro has been processed, register 1 points to a list of fullwords that contain the addresses of ABC and DEF.

The called program must be in virtual storage when the CALL macro is processed. That program may be loaded into virtual storage in one of two ways:

- As part of the program issuing the CALL macro.

  The macro must specify an entry point by a symbolic name. The linkage editor includes the module with that entry point in the phase that issues the CALL macro. As a result, the called program resides in storage together with and as long as the calling program.
- As the phase specified by a LOAD macro.

  The CALL macro specifies register 15 (the entry-point register) into which the entry-point address of the program to be called was loaded. The LOAD macro must precede the CALL for that program.

  Specifying register 15 preceded by a LOAD macro is useful when the same program is called several times while the calling program is running, but is not needed in storage during all of that time.

The SAVE Macro

The macro stores the contents of specified registers in the save area provided by the calling program. Code this macro in the called program before any registers can be modified by this program, preferably at the entry point.

The RETURN Macro

The macro restores the registers whose contents were saved and returns control to the calling program. You can also specify a return code in the macro. Before your program issues the macro, register 13 must contain the address of the save area of the program to which control is to be returned.

```
Code within the calling routine:

   (1)            LA        13,SAVAREA1
                   ...
   (2)            CALL      SUBROUT,(PAR1,PAR2)
  (11)            C         12,ZERO
                   ...
         SAVAREA1  DS        9D
                   ...
         PAR1      DC        C'ABCDEF'
         PAR2      DS        F
         ZERO      DC        F'0'

Code within the called routine:

   (3)  SUBROUT   SAVE      (14,11)
   (4)            BALR      ...
   (4)            USING     ...
   (5)            ST        13,SAVAREA2+4
   (6)            LA        13,SAVAREA2
                   ...
                   ...  Processing
                   ...
   (7)            L         12,RESULT
   (8)            L         13,SAVAREA2+4
   (9)            RETURN    (14,11)
                   ...
  (10)  SAVAREA2  DS        9D
                   ...
```

(1)
Points to the save area in the calling program.

(2)
Passes the parameters PAR1 and PAR2.

(3)
Saves the contents of the registers for the calling program.

(4)
Establishes addressability.

(5)
Provides a back pointer to the calling program's save area.

(6)
Points to the new save area (for tracing purposes).

(7)
Stores the processing results by loading it into register 12.

(8)
Restores the calling program's register from the save area.

(9)
Restores the specified registers and returns control to the instruction at (11).

(10)
The area may be smaller if no other program is called.

(11)
Compares, with 0, the processing result passed by the called program.

Figure 46. Example for Using the Macros CALL, SAVE, and RETURN

Loading a Forms-Control Buffer

Your application might require a change of forms one or more times while it runs. The LFCB macro helps you control a change of forms.

The macro retrieves a control-buffer image stored in an accessible sublibrary as a phase. It loads this image into the printer's forms-control buffer (FCB).

This image corresponds to the layout of the desired output form; it controls the skipping of lines. For information about the contents and format of an FCB image, see z/VSE System Control Statements.

An FCB whose contents has been changed by this macro retains its new contents until one of the following occurs:

- Another LFCB macro is issued for the printer.
- An LFCB command is issued for the printer.
- The SYSBUFLD program is run to reload the printer's FCB.
- A new startup of the system takes place.

It might happen, for example, that you do not know which of the available printers is used when your program runs. The coding example in Figure 47 on page 117 provides for an FCB image to be loaded for a non-PRT1-type printer or a PRT1-type printer.

```
(1)           LFCB    SYSLST,FCBSTD,LPI=8
              LTR     15,15
              BZ      CONTINUE
              CH      15,=H'4'
              BE      TRYAGAIN
              PDUMP   INAREA,OUTAREA
              B       CONTINUE
(2) TRYAGAIN  LFCB    SYSLST,FCBPRT1
              LTR     15,15
              BZ      CONTINUE
              B       CANCL
    CONTINUE  ...
              EOJ
    CANCL     CANCEL  ALL
              ...
```

(1)

If a non-PRT1 printer is being used, the system requests the operator to set the hardware switch for eight lines per inch. If a PRT1 printer is being used, the value 8 in LPI=8 does not match the expected lines-per-inch setting of the image in the first byte. The load request fails with a return code of X'04' in register 15.

(2)

Causes an FCB image for a PRT1-type printer to be loaded if a printer of that type is being used.

Figure 47. Example for Loading an Alternate FCB

The LFCB macro can be useful also in an abnormal-end routine that you specify in an STXIT macro. If, for example, the routine requests a dump of storage to be printed on an IBM 3211 on which indexing is being used, a certain number of characters may be lost on every line of the printed dump. To avoid losing characters, your abnormal-end routine must, before it requests the dump, issue an LFCB macro that specifies an image without an indexing byte.

The LFCB macro generates messages to request operator action (such as changing forms), if manual action is required, and to inform the operator that the FCB of the specified printer has been reloaded.

If the loading of the FCB fails, the system informs your program by a return code in register 15. In your program, examine the return code and take appropriate action. For a list of the return codes, see z/VSE System Macros Reference.

Loading of a printer's FCB by way of the LFCB macro is of advantage, if this printer is an IBM 4248 operating in native mode. If you want to make use of the printer's horizontal-copy function, this function is enabled by an LFCB macro loading a suitable FCB image. Your program need not issue a CNTRL macro to explicitly enable the horizontal copy function.

Multitasking Functions

This section discusses multitasking aspects such as I/O considerations, starting and ending a subtask, sharing and protecting a resource. You find some of the macros that are used for these multitasking functions in the sample program shown in Figure 55 on page 129.

Subtasking and I/O Requests

If several subtasks access the same tape or unit record device, or the same file on a DASD device, the following restrictions have to be observed:

- For tapes:

  Only one OPEN is allowed per device. Therefore, all subtasks must use the same file declarative (DTFxx) macro. In addition, you must ensure that:

  - Only one subtask issues an I/O request at a time.
  - Only one I/O area is specified in the DTFxx macro.
  - No I/O is attempted when a subroutine passes control to an asynchronous user exit such as an IT or OC routine.
- For other devices several subtasks (including user-exit routines) can issue I/O requests at the same time, if each subtask:
  - Uses its own DTFxx macro.
  - Uses its I/O and work areas.
  - Does not share a non-reentrant logic module.

Note: If the extended buffering support for IBM 3800/3200 printers is used, do not attempt an I/O request out of an asynchronous user exit. This may cause an abnormal end of the task.

Starting (Attaching) a Subtask

Depending on the NPARTS specification of the IPL SYS command, up to 512 subtasks may be active in the system at any one time, up to 31 within a partition.

The program page with the entry address of the subtask must be in virtual storage for the attach-subtask request to be successful. The set of program instructions that make up the subtask can be part of one large control section (CSECT) which, possibly, includes also the main task. A subtask can also be a separate phase. In that case, the phase must be read into storage with the LOAD or CDLOAD macro before the ATTACH macro is issued.

Figure 55 on page 129 includes an example of attaching a subtask.

Required Save Areas

The system provides a save area for the main task. For discussion of this area, see “Register Save Area” on page 114.

The attaching task must provide a save area for the subtask it attaches. The address of this area is specified in the SAVE=save area operand of the ATTACH macro. When, later on during processing, the

subtask is interrupted, the system saves in that area the subtask's interrupt status information, the contents of the general registers, and the floating-point registers.

The first eight bytes are reserved for the name of the subtask that is to be attached. The attaching task should insert this name in those eight bytes. The system uses this name to identify the subtask in an abnormal-end message, should one occur.

A second save area (ABSAVE=absave area) is needed if the attached task uses the attaching task's abnormal-end routine. The save area of the attaching task is then reserved for the abnormal-end of only the attaching task.

Testing for Successful Attachment

The request to attach a subtask may not be successful. The maximum number of attachable subtasks may already have been activated, for example. The attaching task must keep control of this.

Register 1 (of the main task) contains the address of an ECB that the system posts when another subtask can be attached again. This register will have the high order bit set to 1 if the attach request was unsuccessful.

Specifying an Event Control Block

In the ATTACH macro, you can specify the name of an event control block (ECB). Provide an ECB (by specifying ECB=ecbname) if other tasks can be affected by an abnormal end of the subtask or if resources are to be controlled by ENQ and DEQ macros within the subtask.

An ECB is a fullword whose format is shown below. Other control blocks can be used as ECBs: CCB and TECB. The format of an ECB is:

Byte
Contents/Meaning

0-1
Reserved.

2
As follows:

Bits
Meaning if Set to 1

0
(X'80') – A program-requested end of the subtask.

0-1
(X'C0') – An abnormal end of the subtask.

2-7
Reserved.

3
Reserved.

Posting is done in bit 0 of byte 2 of these blocks. However, do not post (with the POST macro) the ECB specified by the ATTACH macro. It causes a task waiting on this ECB to be set ready-to-run, although the posting subtask has not yet ended. For task synchronization use a second ECB which can then be used by the POST macro.

A task may wait for a subtask to end by issuing a WAIT or WAITM macro with the ECB as an operand. Use a WAIT macro if your task is to wait for a single event to occur; use a WAITM macro if your task can continue processing when just one out of a number of events occurred. When the subtask ends, the system posts the ECB.

The end of a task is considered to be abnormal if it does not result from one of the following macros issued by the program itself: CANCEL, DETACH, DUMP, JDUMP, and EOJ.

The type of termination is determined by the condition encountered last. If an AB exit routine was set up for the subtask, it normally ends with one of the above macros, and the abnormal-end bit is not set by the system. In this case, your program can communicate the abnormal condition by setting the bit itself or by any other means.

Changing the Processing Priority

If an attach request is successful, the attached subtask gets higher priority than the main task. Among all the subtasks in a partition, a subtask can give itself the lowest processing priority of all subtasks attached in the partition by issuing the CHAP macro.

Ending (Detaching) a Subtask

A subtask is normally ended by a DETACH macro issued by the main task or by the subtask itself. A subtask can detach itself also by:

- One of the macros CANCEL, EOJ, DUMP, and JDUMP.

  A CANCEL issued in a subtask detaches only the subtask, except if ALL was specified in the macro. CANCEL ALL in a subtask causes all processing in the partition to be canceled. For a discussion of the macros DUMP and JDUMP, see the section “Requesting Storage Dumps” on page 132.

  The EOJ macro issued by a subtask ends only this subtask. All other tasks in the same partition continue processing.

When a subtask is detached, the system ensures that all pending I/O operations are completed and any tracks held by this subtask are freed. If the subtask has an ECB, that ECB is posted, and any tasks waiting on the ECB are removed from the wait state. The task with the highest priority then gets control.

Figure 55 on page 129 includes an example of detaching subtasks. The main task attaches two subtasks: SUBTASK1 and SUBTASK2. When SUBTASK1 completes processing, it sets on indicator ESUB1 in the main task. The main task then detaches SUBTASK2 by issuing a DETACH macro and specifying the save area of SUBTASK2. When SUBTASK1 completes its processing, it detaches itself.

Task-to-Task Communication within Partition

Tasks can communicate with each other through event control blocks (ECBs) as described under “Specifying an Event Control Block” on page 119. A task sets itself into the wait state by issuing a WAIT macro that specifies an ECB. To release that task from the wait state, another task must issue a POST macro that specifies the same ECB.

The task that issues the WAIT macro remains in the wait state until the specified ECB is posted.

Note:

1. A task never regains control if it is waiting for a CCB to be posted by another task's I/O completion.
2. Do not use a telecommunication ECB or an RCB to wait on because bit 0 of byte 2 of these blocks never gets posted.

When control returns to a task that was waiting for one of a number of events to occur (WAITM), register 1 points to the posted ECB. This allows the task to find out which event removed it from the wait state.

If WAITM was issued in 31-bit mode, register 1 holds the entry of the ECB list; if register 1 holds the last entry, the high-order bit is set (only for user-defined ECB lists).

A task that issues the WAITM macro should include code for an escape if an event might not occur. Such a condition could arise if, for example, the task that is to post an event ends abnormally.

In Figure 48 on page 121, the WAITM macro specifies a preferred event (ECBPREF) as the first operand and a secondary event (ECBSEC) as the second operand. The preferred event is the successful completion of SUBTASK1, which is indicated by

```
POST ECBPREF.
```

If the subtask is terminated before it can finish its processing, the supervisor posts the ECB defined as the secondary event by:

```
ATTACH...,ECB=ECBSEC
```

For either event, the address of the posted ECB is in register 1 after the WAITM macro has been issued. This address allows you, for example, to select a certain routine in your program. In the example of Figure 48 on page 121, a branch instruction points to a list of ECBs. For each ECB, the list provides a branch instruction to the routine that is to receive control when that ECB is posted. The list may include up to 16 ECBs.

```
MAINTASK    BALR   12,0
            USING  *,12
             ...
            ATTACH SUBTASK1,SAVE=SAVE1,ECB=ECBSEC
             ...
            LA     4,ECBSEC
            WAITM  ECBPREF,ECBSEC  WAIT FOR PREFERRED OR SECONDARY
*                                  EVENT
            B      4(1)      BRANCH INTO A VECTOR TABLE
             ...
PREVENT     EQU    *         CONTINUE AFTER PREFERRED EVENT
             ...
SEVENT      EQU    *         CONTINUE AFTER SECONDARY EVENT
             ...
            EOJ              MAIN TASK END OF JOB
             ...
SUBTASK1    EQU    *
             ...
            POST   ECBPREF   POST COMPLETION OF PREFERRED EVENT
             ...
ECBSEC      DC     F'0'      ECB FOR SECONDARY EVENT
            B      SEVENT    VECTOR BRANCH FOR SECONDARY EVENT
ECBPREF     DC     F'0'      ECB FOR PREFERRED EVENT
            B      PREVENT   VECTOR BRANCH FOR PREFERRED EVENT
```

Figure 48. Waiting for Preferred and Secondary Events

When a task posts an ECB, any task waiting on this ECB to be posted is removed from the wait state.

You can code your program to have just one task or all tasks waiting on a certain event removed from the wait state. To have all tasks removed, simply issue a POST macro with the ECB name specified as the only operand. Example:

```
POST   ST1ECB
```

To have only one task removed, specify also the name of that task's save area. Example:

```
POST   ST1ECB,SAVE=ST1SAVE
```

Specifying the SAVE operand saves time. The operand ensures that only the subtask owning the specified save area is taken out of the wait state.

Note: Do not use this technique if the ECB to be posted is the one specified in the ATTACH macro and the macros ENQ and DEQ are used. The DEQ macro removes from the wait state all tasks waiting for the protected resource. Instead, use two different ECBs. However, your program is responsible for resetting the traffic bit (bit 0 of byte 2) in the second ECB. To do this, use the instruction

```
MVI ecbname+2,X'00'
```

Figure 49 on page 122 shows how to use the POST macro. In the example, the subtask SUBTASK1 depends on input from either of the subtasks SUBTASK2 and SUBTASK3. Therefore:

1. SUBTASK1 issues a WAITM macro on the ECBs for those subtasks.
2. The WAITM macro places SUBTASK1 into the wait state.
3. Control passes to SUBTASK2 and then to SUBTASK3.

4. When either of the two subtasks has the input for SUBTASK1, it posts its ECB. This removes SUBTASK1 from the wait state.

When SUBTASK1 finishes processing, it posts its ECB, thus causing the main task to be taken out of the wait state. The main task can then detach SUBTASK1.

```
MAINTASK    BALR    12,0
            USING   *,12
             ...
            ATTACH  SUBTASK1,SAVE=ST1SAVE,ECB=ST1ECBS
             ...
            ATTACH  SUBTASK2,SAVE=ST2SAVE,ECB=ST2ECBS
             ...
            ATTACH  SUBTASK3,SAVE=ST3SAVE,ECB=ST3ECBS
             ...
            WAIT    ST1ECB          WAIT FOR COMPLETION OF SUBTASK 1
            DETACH  SAVE=ST1SAVE    DETACH SUBTASK 1
             ...
            EOJ
SUBTASK1    ST      1,MTSVADR1      STORE THE ADDRESS OF THE MAIN TASK
*                                   SAVE AREA
             ...
            WAITM   ST2ECB,ST3ECB   WAIT FOR SUBTASK 2 OR SUBTASK 3
             ...
ST1EOJ      L       0,MTSVADR1      GET THE ADDRESS OF THE MAIN TASK
*                                   SAVE AREA
            POST    ST1ECB,SAVE=(0)    POST THE ECB FOR THE MAIN TASK
            WAIT    ECB1A           WAIT TO BE DETACHED FROM THE MAIN
*                                   TASK
             ...
SUBTASK2    EQU     *
             ...
ST2A        EQU     *
             ...
            POST    ST2ECB          POST ECB FOR SUBTASK 1
             ...
            B       ST2A
             ...
SUBTASK3    EQU     *
             ...
ST3A        EQU     *
             ...
            POST    ST3ECB          POST ECB FOR SUBTASK 1
             ...
            B       ST3A
             ...
MTSVADR1    DC      F'0'            SAVE AREA ADDRESS FOR THE MAIN TASK
ECB1A       DC      F'0'            DUMMY ECB FOR SUBTASK 1
ST1ECB      DC      F'0'            ECB FOR SUBTASK 1
ST2ECB      DC      F'0'            ECB FOR SUBTASK 2
ST3ECB      DC      F'0'            ECB FOR SUBTASK 3
ST1ECBS     DC      F'0'            ATTACH ECB FOR SUBTASK 1
ST2ECBS     DC      F'0'            ATTACH ECB FOR SUBTASK 2
ST3ECBS     DC      F'0'            ATTACH ECB FOR SUBTASK 3
```

Figure 49. Use of the POST Macro

Resource Protection

When two or more tasks in the same partition access the same resource (certain data for updating, for example), protection is required to avoid that the resource is used concurrently by these tasks. Such protection is possible if every task in the partition uses the RCB, ENQ, and DEQ macros. It is not possible, however, for system units such as the SYSLST device.

A task protects a resource by issuing an ENQ (enqueue) macro that refers to the resource by the name of an eight-byte resource control block (RCB). The format and content of an RCB is as follows:

Bytes
Contents/Meaning

0
All bits are set to 1 to indicate that the resource has been placed in a priority queue by the ENQ macro.

1-3
Reserved

4-7
ECB address of current resource owner (4-byte address). Bit 0 of byte 4 will be set to 1 when another task is waiting to use the resource.

A subtask that enqueues a resource must have an ECB specified in its ATTACH macro. Do not use this ECB for any purpose other than resource protection as long as the resource is enqueued. The address of the ECB is stored in the RCB as shown above.

A task requesting the use of a resource is enqueued and receives control if the resource is free. The task is put into the wait state if the resource has already been enqueued by another task. If an ENQ macro is issued for an already enqueued resource, the system indicates this in the RCB and stores the address of the current resource owner's ECB in register 1 of the task that is placed into the wait state.

Once a resource has been enqueued by a task, only this task can dequeue the resource. The task does this by issuing the DEQ macro.

If other tasks are enqueued on the same RCB, the DEQ macro frees, from their wait condition, all other tasks that were waiting for that resource. The task with the highest priority then gets control. If no other tasks are waiting for the RCB, control returns to the dequeuing task.

Figure 50 on page 123 shows a main task with two subtasks that use the same file and protect this file from simultaneous access. The subtasks access the file in a common subroutine. This subroutine is not reentrant, and therefore, the file cannot be accessed with track hold. Each subtask must enqueue the RCB associated with the file and dequeue it when the file can be released.

```
MAINTASK   BALR    12,0
           USING   *,12
            ...
SUBTASK1   EQU     *
            ...
SBTASK1A   ENQ     RCB1        PROTECT THE RESOURCE
           BAL     4,WRITEDTA  WRITE A RECORD
           DEQ     RCB1        RELEASE THE RESOURCE
            ...
           B       SBTASK1A
            ...
SUBTASK2   EQU     *
            ...
SBTASK2A   ENQ     RCB1        PROTECT THE RESOURCE
           BAL     4,WRITEDTA  WRITE A RECORD
           DEQ     RCB1        RELEASE THE RESOURCE
            ...
           B       SBTASK2A
            ...
WRITEDTDA  EQU     *
            ...
RCB1       RCB                 RESOURCE CONTROL BLOCK
*                              FOR THE ROUTINE WRITEDTA
```

Figure 50. Sharing a Resource in a Common Subroutine

In Figure 51 on page 124, two subtasks share a routine named TOTAL. This routine, which is part of the first subtask (SUBTASK1) is protected by the RCB named RCBA. This protection works only if every segment of code within the partition does the following when it uses TOTAL:

1. Issues an ENQ macro for RCBA before the branch to the routine.
2. Issues a DEQ macro for RCBA when having finished using the routine.

This is accomplished by branching to the same code in subtask 1.

The common code need not be reentrant. Ensure, however, that the values for constants associated with the subroutine do not have to be retained from one use of the common routine to the next. If any values must be retained, save them in the using subtask and restore them when required.

```
MAINTASK    START  0
             ...
            ATTACH SUBTASK1,SAVE=ST1SAVE,ECB=ST1ECB
             ...
            ATTACH SUBTASK2,SAVE=ST2SAVE,ECB=ST2ECB
SUBTASK1     ...
ENQRSRCE    ENQ    RCBA        PROTECT RESOURCE TOTAL
TOTAL       EQU *              USED BY BOTH SUBTASK 1 AND
*                              SUBTASK 2
             ...
            DEQ    RCBA        RELEASE RESOURCE TOTAL
             ...
SUBTASK2    EQU    *
             ...
            B      ENQRSRCE    PROCESS TOTAL
             ...
RCBA        RCB                RCB FOR RESOURCE TOTAL
             ...
```

Figure 51. Sharing a Routine Which Is Part of One Task

In Figure 52 on page 124, the subtasks again use the same resource, but they access the resource from code of their own. The resource, called RESRCA, may be a data area for example or a file defined by a DTFxx macro. Whatever the resource is, RESRCA is protected from being used by subtask 2 while it is used by subtask 1, and vice versa. Thus, if all tasks enqueue and dequeue each access to RESRCA, the resource is protected against access from any other task while it is being used by one task.

This protection works if the resource as a whole is in storage. However, if the resource is a file, only the data being operated upon is protected while in storage; the file itself on on an external storage device is not necessarily protected. If the file is on disk, use the track hold function, which is discussed under “DASD Record Protection (Track Hold)” on page 126.

```
MAINTASK  START 0
           ...
          ATTACH SUBTASK1,SAVE=ST1SAVE,ECB=ST1ECB
           ...
          ATTACH SUBTASK2,SAVE=ST2SAVE,ECB=ST2ECB
           ...
SUBTASK1  EQU     *
           ...
          ENQ     RCBA           PROTECT RESOURCE RESRCA
          GET     RESRCA,WKAREA
           ...
          DEQ     RCBA           RELEASE RESOURCE RESRCA
           ...
SUBTASK2  EQU     *
           ...
          ENQ     RCBA           PROTECT RESOURCE RESRCA
          PUT     RESRCA,WKAREA
           ...
          DEQ     RCBA           RELEASE RESOURCE RESRCA
           ...
RCBA      RCB                    RCB FOR RESOURCE RESRCA
RESRCA    DTFSD ...              SHARED RESOURCE
```

Figure 52. Sharing a Resource in Different Subroutines

In your program design, be careful to avoid conditions that might result in a deadlock situation. Consider the following two segments of code being executed concurrently by two tasks:

```
  Task1 Executes        Task2 Executes

  A   ENQ  RCBA         B   ENQ  RCBB
       ...                   ...
  C   ENQ  RCBB         D   ENQ  RCBA
       ...                   ...
      DEQ  RCBA             DEQ  RCBB
```

If the macros happen to be processed in the sequence A, B, C, then both tasks end at statements C and D without a chance of ever regaining control.

Resource-Share Control

Another set of macros protects a resource against concurrent use by different tasks in the same or in other partitions, on the same or on a different system. This set of macros provides for controlled sharing of the resource. The macros:

- Define a protected resource: the macros DTL, GENDTL, MODDTL
- Control access to the resource: the macros LOCK, UNLOCK.

Protection is possible only if all programs accessing the protected resource use the protection service in a consistent manner. These programs must use the macros and adhere to the naming conventions established for the shared resource.

Defining a Sharable Resource

A resource to be protected by this share-control service must be defined in a lock control block called DTL (define the lock). The DTL macro is used to assemble a DTL in your program; the GENDTL macro is used to dynamically build a DTL while your program is running.

The DTL indicates the locally defined name of the protected resource and how access to the resource may be shared with other programs. In the macro that defines the DTL, you can specify:

- The type of control: either E (exclusive) or S (sharable).
- To what extent the resource may be shared.
- The scope of share control, namely whether a locked resource:
  - Is released automatically at the end of program execution.
  - Remains locked for the next job step (the resource is always released at the end of a job).
  - May be unlocked only by the same task or also by another task.

Controlling a Sharable Resource

Figure 53 on page 125 illustrates how the resource share control macros might be used.

```
                ...
                LOCK        MYDTL
                ...
                MODDTL      ADDR=MYDTL,CONTROL=S,CHANGE=ON
(1)             UNLOCK      MYDTL
                ...
                MODDTL      ADDR=MYDTL,CHANGE=OFF
(2)             UNLOCK      MYDTL
                ...
(3)   MYDTL     DTL         NAME=RESOURCE,CONTROL=E,LOCKOPT=1
```

(1)

The resource is not unlocked. It remains locked, but may now be shared with other tasks.

(2)

Because CHANGE=OFF is specified in the preceding MODDTL macro, the resource is actually unlocked.

(3)

CONTROL=E with LOCKOPT=1 indicates that no other task can gain access to the resource RESOURCE as long as access is controlled by this DTL.

Figure 53. Example of the UNLOCK Macro

Once the DTL for a resource exists, a task can request control over this resource with the LOCK macro and give up control over the resource with the UNLOCK macro. The system maintains a lock-request count which reflects the number of lock requests issued for the resource.

When a LOCK request is issued and the resource is already locked by another task, further system action depends on your specification in the FAIL operand of the LOCK macro. For example, you can specify that control is to return to the requesting task or that the requesting task be set into the wait state until a locked resource is unlocked. If control is to be returned, your program must test the return code set by the system to determine the action to be taken next.

A task or partition may issue one or more lock requests for a resource. To yield control over a resource completely, a task or partition must, for this resource, issue as many unlock requests as it has issued lock requests.

The MODDTL macro modifies a lock control block while the program is running. This is the normal function of the macro. In addition, the macro can be used to lower the level of lock control over a locked resource.

If the MODDTL macro specifies CHANGE=ON, the the macro causes the next UNLOCK macro for the resource to keep the resource locked, but with a lower locking level. Another task waiting for this resource can be dispatched again. This method of reducing the lock level can be employed if the lock level is defined with the most stringent values: CONTROL=E (exclusive) and LOCKOPT=1.

DASD Record Protection (Track Hold)

While a record is being modified by one task, it must be prevented from being accessed by another task. VSE includes the DASD record protection support (also called "track-hold function") to ensure this kind of data integrity. This support is available for both CKD disks and FBA disks.

For a CKD disk, the unit of data transfer is one block (or a physical record); for an FBA device, this unit is a control interval. For ease of reading, the data transfer unit is referred to in this section as "track".

Prerequisites

The track-hold support is available if:

1. The supervisor of your system was generated with the TRKHLD operand specified in the FOPT macro.
2. Any task that accesses data on disk refers to a DTFxx macro with HOLD=YES.
3. Your program makes a separate register save area available. Your program must load the address of this area into register 13 before it issues any READ or WRITE macro.

If these requirements are met, the function provides DASD record protection for your programs.

Scope of the Support

The track-hold function can be used for updating a DTFSD file and for processing a DTFSD work file or a DTFDA file.

Updating a DTFSD File

The track being held is freed automatically by the system. More specifically, the next GET issued to a new track for the file frees the previous hold. Your program need not issue the FREE macro.

Processing a DTFSD Work File or a DTFDA File

Your program must issue the FREE macro for each hold that is placed on the track. The system places a hold on a track each time the track is accessed with a READ or WRITE macro, and each hold is released by issuing the FREE macro.

Hints for Programming

A pending hold is released automatically when your program issues a CLOSE macro for the file or a DETACH macro for the accessing task.

The maximum number of **different tracks** that may be held at a time is specified for supervisor generation. If a task attempts to exceed the generated limit, this task is placed into the wait state until one of the tracks currently held is freed. If one task holds as many tracks as can be held and attempts to issue a hold for an additional track, this task is canceled by the system.

The **same track** can be held more than once without an intervening FREE if the hold requests are from the same task. However, the same number of FREE macros must be issued before the track is completely freed. A task is canceled if it issues more than 16 hold requests without an intervening FREE or if it issues a FREE for a file that does not have a hold request for a track.

If a task requests a hold for a track that is being held by another task, the requesting task is placed into the wait state at the requesting (GET or WAITF) macro. The request is fulfilled after the track is freed and when control returns to the requesting task.

If it holds more than one track, your program may inadvertently put the entire system into the wait state. This occurs if each task is waiting for a track that is already held by another task. You can avoid this by a FREE for each track held by a task before this task places (or attempts to place) a hold on another track.

Coding Example

Figure 54 on page 128 shows an example of the use of the track hold function in a multitasking program.

Although track hold works across partitions, the example shows only two subtasks sharing the same (DTFDA defined) file of data. A similar set of routines could be executing in another partition and share the file with this partition.

HOLD=YES must be specified in both DTFDAs. If register 13 is not altered between I/O operations performed by a given task, it needs to be initialized only once. If other reentrant access methods were used by the subtask in addition, register 13 would have to be initialized for each I/O request.

```
MAINTASK   START    0
            ...
           ATTACH   SUBTASK1,SAVE=ST1SAVE,ECB=ST1ECB
            ...
           ATTACH   SUBTASK2,SAVE=ST2SAVE,ECB=ST2ECB
            ...
SUBTASK1   OPEN     DAFILE1                  Open the file DAFILE1
            ...
           LA       13,DASAVE1               Load the address of the
            ...                              save area DASAVE1 into
            ...                              register 13
           READ     DAFILE1,KEY              Read the record and hold
            ...                              the track of the record
           WAITF    DAFILE1
            ...
           WRITE    DAFILE1,KEY              Write the updated
           WAITF    DAFILE1                  record
           FREE     DAFILE1                  Release the track
            ...
SUBTASK2   OPEN     DAFILE2                  Open the file DAFILE2
            ...
           LA       13,DASAVE2               Load the address of the
            ...                              save area DASAVE2 into
            ...                              register 13
           READ     DAFILE2,KEY              Read the record and hold
           WAITF    DAFILE2                  the track of the record
            ...
           WRITE    DAFILE2,KEY              Write the updated record
           WAITF    DAFILE2
            ...
           FREE     DAFILE2                  Release the track
            ...
DAFILE1    DTFDA    HOLD=YES,...
            ...
DAFILE2    DTFDA    HOLD=YES,...
            ...
DASAVE1    DC       8D'0'                    Save areas used for
DASAVE2    DC       8D'0'                    shared and reentrant
                                             modules
```

Figure 54. Using the Track Hold Facility

Shared Modules and Files

The DTFxx and xxMOD macros for the various file types include the operand RDONLY=YES. This operand indicates that a sharable read-only module is to be generated.

Each time a read-only module is entered, register 13 must contain the address of a 72-byte, doubleword-aligned save area. A separate save area is needed in addition if an exit to a user routine (AB, IT, OC, PC) exists and the exit routine issues I/O request(s) requiring the same logic module as the main routine. Each task using a read-only module requires its own unique save area in addition to the save areas that might be needed for multitasking and program linkage.

If an ERROPT or a WLRERR routine issues I/O macros, using the same read-only module that passed control to the routine, your program must provide yet another save area: one save area for the initial I/O and the other for each I/O request from the ERROPT or WLRERR routine. Before control returns to the module that entered the ERROPT routine, register 13 must contain the address of the save area originally specified for the task.

If several tasks are to share one data file, use a reentrant module. In addition, provide a DTF table for the file in each task, unless you use the ENQ and DEQ macros. Each task can either open its own DTF, or the main task in the partition can open all files for the subtasks.

There are two methods that can be used for a shared file:

1. Supply a separate set of label statements (DLBL and EXTENT, etc.) for each corresponding DTFxx-defined file,
2. Assemble each DTFxx macro and program (subtask) separately with the same file name and provide one set of label statements.

In case “2” on page 128, each separately assembled program must open its DTFxx file.

Multitasking Sample Program

Figure 55. Multitasking Sample Program

```
******************   MAIN PROGRAM   *********************************
* REGISTER USAGE: R12 = BASE REGISTER                                 *
*                 R5  = MESSAGE ADDRESS                               *
*                 R10 = BRANCH AND LINK REGISTER                      *
* DO NOT USE REGISTER 2 AS BASE REGISTER IF YOU ISSUE A -PUTR- MACRO *
***********************************************************************
MAINTASK START X'78'
         PRINT NOGEN
         BALR  R12,0
BASEADDR EQU   *
         USING BASEADDR,R12
         OPEN  CONSIOA       OPEN CONSOLE INPUT/OUTPUT FILE
         BAL   R10,ABXITM    ESTABLISH ABNORMAL END EXIT FOR MAIN TASK
         BAL   R10,ATTST1    ATTACH SUBTASK1
         BAL   R10,ATTST2    ATTACH SUBTASK2
MPOST1   POST  ST1ECBM       GIVE CONTROL TO SUBTASK1
         WAIT  ST1ECB        WAIT FOR POSTING FROM SUBTASK1 (SEE *1)
*1 THIS WAIT ECB (ST1ECB) WILL BE RELEASED BY THE DETACH MACRO
*1 ISSUED IN SUBTASK1.
************  STATEMENTS OF YOUR PROGRAM ****************
MT40     EQU   *
         WAIT  MTECB2        WAIT FOR POSTING FROM SUBTASK2
         DETACH SAVE=ST2SAVE DETACH SUBTASK2 (SAVE=  IS MANDATORY)
         LA    R5,MSGEND     ADDRESS OF END MESSAGE ==> R5
         BAL   R10,PUTCONS   OUTPUT OF END MESSAGE ON CONSOLE
         LA    R15,0         SET RETURN CODE FOR EOJ
CLOSEALL CLOSE CONSIOA       CLOSE ALL FILES STILL OPEN
         EOJ   RC=(15)       EOJ - RETURN CODE IN R15
***********************************************************************
ABMAIN   EQU   *             ENTERED IF ABNORMAL END OF MAIN TASK
         LA    R5,MSGABM     ADDRESS OF ABNORMAL END MESSAGE
         BAL   R10,PUTCONS   OUTPUT OF ABNORMAL END MESSAGE
         LA    R15,8         SET RETURN CODE FOR EOJ
         B     CLOSEALL
         SPACE 2
ST1ABEND EQU   *
         L     R1,SUB2ECB
         POST  (1)            POST ECB OF SUBTASK2
         JDUMP
         SPACE 2
ST2ABEND EQU   *
         L     R1,SUB1ECB
         POST  (1)            POST ECB OF SUBTASK1
         POST  MTECB2         POST THE MAIN TASK'S ECB
         JDUMP
****** DEFINITION AREA FOR MAIN TASK **********************************
MSGEND   DC    CL80'MAIN NORMAL ENDED'
MSGABM   DC    CL80'MAIN ABNORMAL ENDED'
         DS    0D
MTABSV   DC    9D'0'         MAIN-TASK ABNORMAL END SAVE AREA
MTECB2   DC    F'0'          MAIN-TASK ECB FOR POST FROM SUBTASK2
ST1ECBM  DC    F'0'          SUBTASK1 ECB FOR POST FROM MAIN TASK
         EJECT
```

```
****** OUTPUT ON CONSOLE - ALPHANUMERIC *******************************
PUTCONS  EQU   *
         ENQ   CONSRCB       PROTECT RESOURCE FOR CONSOLE OUTPUT
         PUT   CONSIOA,(R5)
         DEQ   CONSRCB       RELEASE RESOURCE FOR CONSOLE OUTPUT
         BR    R10
****** FILE DEFINITION AREA ***************************************
CONSIOA  DTFCN                                                  X
               DEVADDR=SYSLOG,                                  X
               IOAREA1=CONINOUT,                                X
               BLKSIZE=80,                                      X
               INPSIZE=80,                                      X
               TYPEFLE=CMBND,                                   X
               RECFORM=FIXUNB,                                  X
               WORKA=YES
CONINOUT DC    CL80' '
               SPACE 2
```

```
*********************************************************************
*        ATTACH SUBTASK 1                                           *
*********************************************************************
ATTST1   EQU   *
         MVC   ST1SAVE(8),ST1NAME SUBTASK1 NAME => ST-SAVEAREA (SEE *2)
*2
*2  THE SUBTASK NAME IS USED FOR IDENTIFICATION IN MESSAGES WRITTEN ON
*2  SYSLOG.
*2
         ATTACH SUBTASK1,ECB=ST1ECB,SAVE=ST1SAVE,ABSAVE=ST1ABSV (SEE*3)
*3
*3  SUBTASK1 = ENTRY POINT OF THE SUBTASK
*3  ST1ECB   = NAME OF SUBTASK'S EVENT CONTROL BLOCK
*3  ST1SAVE  = NAME OF SUBTASK'S SAVE AREA
*3  ST1ABSV  = NAME OF SUBTASK'S ABNORMAL END SAVE AREA
*3
         LTR   R1,R1         TEST IF ATTACH IS SUCCESSFUL (SEE *4)
*4
*4  IF THE ATTACH WAS SUCCESSFUL, THE MAIN TASK STORES THE ENDING
*4  ADDRESS OF SUBTASK'S SAVE AREA IN REGISTER 0 FOR LATER REFERENCE
*4  AND THE ADDRESS OF THE ATTACHING TASK'S SAVE AREA IN REGISTER 1.
*4
         BNM   ATTST10       BRANCH IF SUCCESSFUL
         WAIT  (1)           WAIT TO RETRY ATTACH
         B     ATTST1        BRANCH TO RETRY
ATTST10  BCTR  R0,R0         GET END OF SAVE AREA SUBTASK1 AND
         ST    R0,ST1SVEND   STORE THE END ADDRESS
         BR    R10           BRANCH AND LINK RETURN VIA REGISTER 10
ST1SAVE  DC    16D'0'        SAVE AREA SUBTASK1 WITH FLOAT REGS
ST1ABSV  DC    9D'0'         AB SAVE AREA SUBTASK1
ST1ECB   DC    F'0'          ECB SUBTASK1
ST1SVEND DC    F'0'          END ADDRESS SUBTASK1 SAVE AREA
ST1NAME  DC    CL8'SUBTASK1' NAME OF SUBTASK1
         EJECT
```

```
*********************************************************************
*        ATTACH SUBTASK 2 (SEE NOTES *2 THROUGH *4 IN PART 3)       *
*********************************************************************
ATTST2   EQU   *
         MVC   ST2SAVE(8),ST2NAME PROVIDE SUBTASK NAME IN ST-SAVE AREA
         ATTACH SUBTASK2,ECB=ST2ECB,SAVE=ST2SAVE,ABSAVE=ST2ABSV
         LTR   R1,R1         TEST IF ATTACH IS SUCCESSFUL
         BNM   ATTST20       BRANCH IF SUCCESSFUL
         WAIT  (1)           WAIT TO RETRY ATTACH
         B     ATTST2        BRANCH TO RETRY
ATTST20  BCTR  R0,R0         GET END OF SAVE AREA SUBTASK2 AND
         ST    R0,ST2SVEND   STORE THE END ADDRESS
         BR    R10           BRANCH AND LINK RETURN VIA REGISTER 10
ST2SAVE  DC    16D'0'        SAVE AREA SUBTASK2 WITH FLOAT REGS
ST2ABSV  DC    9D'0'         AB SAVE AREA SUBTASK2
ST2ECB   DC    F'0'          ECB SUBTASK2
ST2SVEND DC    F'0'          END ADDRESS SUBTASK2 SAVE AREA
ST2NAME  DC    CL8'SUBTASK2' NAME OF SUBTASK2
         EJECT
*  ESTABLISH ABNORMAL END EXIT
ABXITM   EQU   *
         STXIT AB,ABEXIT,MTABSV,OPTION=NODUMP
         BR    R10
*  ABNORMAL EXIT ROUTINE REFERENCED BY STXIT MACRO
ABEXIT   EQU   *
         BALR  R12,0
         USING *,R12         ESTABLISH ADDRESSABILITY TO ABASE PTR
         L     R12,ABASE
         USING BASEADDR,R12  ESTABLISH ADDRESSABILITY TO BASE ADDR
*   SAVE CANCEL CODE LOCATED IN REGISTER 0, IF NECESSARY
         LR    R6,R1         SAVE AB EXIT SAVE AREA ADDRESS
*   R0, R1 WILL BE DESTROYED BY DEQ
         DEQ   CONSRCB     RELEASE RESOURCE FOR CONSOLE OUTPUT (SEE *5)
*5
*5 DEQ WILL BE IGNORED, IF RESOURCE CONSRCB IS NOT PROTECTED.
*5 WHEN AN ABNORMAL TERMINATION CONDITION OCCURS DURING THE PUT
*5 PROCESS, RESOURCE CONSRCB WILL BE RELEASED.
*5
         C     R6,=A(ST1ABSV) SUBTASK1 ABNORMALLY ENDED?
         BE    ST1ABEND       IF YES GO TO ST1ABEND
         C     R6,=A(ST2ABSV) SUBTASK2 ABNORMALLY ENDED?
         BE    ST2ABEND       IF YES GO TO ST2ABEND
         EXIT  AB             MUST BE MAIN TASK
         B     ABMAIN         CONTINUE MAIN-TASK PROCESSING
```

```
ABASE    DC A(BASEADDR)       MAIN TASK BASE ADDRESS
SUB1ECB  DC A(ST1ECB20)       ECB ADDRESS WITHIN SUBTASK1
SUB2ECB  DC A(ST2ECB10)       ECB ADDRESS WITHIN SUBTASK2
CONSRCB  RCB                  RESOURCE CONTROL BLOCK FOR CONSOLE OUTPUT
         EJECT
```

```
****** REGISTER EQUIVALENTS ****************************************
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         SPACE 2
*********************************************************************
*          S U B T A S K  1                                         *
*********************************************************************
         CNOP  0,4
SUBTASK1 EQU   *      SUBTASK1 = ENTRY POINT GIVEN BY ATTACH MACRO
         BALR  R3,0   (SEE *6)
         USING *,R3   (SEE *6)
*6
*6  THE MAINTASK AND THE SUBTASK MAY USE DIFFERENT BASE REGISTERS.
*6  THIS IS NOT NECESSARY IF ADDRESSABILITY IS ENSURED BY
*6  THE MAINTASK'S BASE REGISTER (R12 IN THIS EXAMPLE).
*6
         LA    R5,MSG101        ADDRESS OF ATTACH MESSAGE ==> R5
         BAL   R10,PUTCONS      OUTPUT OF MESSAGE TO CONSOLE
         WAIT  ST1ECBM          WAIT TO BE POSTED FROM MAIN TASK
         MVI   ST1ECBM+2,X'00'  RESET ST1ECBM IN WAIT STATE
         POST  ST2ECB10         RELEASE WAIT STATE OF SUBTASK2 ECB10
*
*   STATEMENTS OF YOUR PROGRAM
*
         WAIT  ST1ECB20         WAIT TO BE POSTED FROM SUBTASK2
         LA    R5,MSG102        ADDRESS OF SUBTASK1 END MESSAGE ==> R5
         BAL   R10,PUTCONS      OUTPUT OF END MESSAGE ON CONSOLE
         DETACH                 SUBTASK1 DETACHES ITSELF
*                                          AND POSTS ST1ECB
         SPACE
         DS    0F               SET FULLWORD BOUNDARY
ST1ECB20 DC    F'0'             ECB TO BE POSTED IN SUBTASK2
MSG101   DC    CL80'SUBTASK1 ATTACHED'
MSG102   DC    CL80'END OF SUBTASK1 REACHED'
         EJECT
```

```
*********************************************************************
*          S U B T A S K  2                                         *
*********************************************************************
         CNOP  0,4
SUBTASK2 EQU   *               ENTRY POINT GIVEN BY ATTACH MACRO
         BALR  R4,0           (SEE *6 ABOVE)
         USING *,R4
         ST    R1,MTSVADR2   STORE ADDR. OF MAIN TASK SAVE AREA (SEE*7)
*7
*7  THE ADDRESS OF THE MAIN TASK'S SAVE AREA IS ONLY NECESSARY FOR THE
*7  POST OPERAND SAVE=(R8) (SEE THE EXAMPLE BELOW).
*7
         LA    R5,MSG201        ADDRESS OF ATTACH MESSAGE ==> R5
         BAL   R10,PUTCONS      OUTPUT OF MESSAGE TO CONSOLE
WAIT2    WAIT  ST2ECB10         WAIT TO BE POSTED FROM SUBTASK1
         MVI   ST2ECB10+2,X'00' RESET ST2ECB10 IN WAIT STATE
*
*   STATEMENTS OF YOUR PROGRAM
*
         LA    R5,MSG202        ADDRESS OF END MESSAGE ==> R5
```

```
         BAL   R10,PUTCONS      OUTPUT OF MESSAGE TO CONSOLE
         POST  ST1ECB20         RELEASE WAIT STATE OF SUBTASK1 ECB20
         L     R8,MTSVADR2      LOAD ADDRESS OF MAIN TASK'S SAVE AREA
         POST  MTECB2,SAVE=(R8) RELEASE WAIT STATE OF MAIN TASK
         B     WAIT2
ST2ECB10 DC    F'0'             ECB TO BE POSTED IN SUBTASK1
MSG201   DC    CL80'SUBTASK2 ATTACHED'
MSG202   DC    CL80'SUBTASK2 WAITING FOR DETACH FROM MAIN'
MTSVADR2 DC    F'0'             ADDRESS OF MAIN TASK SAVE AREA
         EJECT
****** DSECT OF ABNORMAL-END SAVE AREA ********************************
         MAPSAVAR            MAPPING OF EXIT SAVE AREA
         END
```

Requesting Storage Dumps

When a program ends abnormally, a dump of the virtual storage areas used by the program at that point in time can help you find the cause. Obtain a printout of this dump or have the dump displayed on a terminal. For a full discussion of storage dumps, z/VSE Diagnosis Tools.

VSE provides several macros to request a dump of storage. In your program, you can use these macros, for example, at the end of an exit routine for handling an abnormal-end condition. Following is a summary of the functions of the macros that request storage dumps:

The DUMP Macro

The macro dumps, in hexadecimal format, the contents of:

- The supervisor area or only some supervisor control blocks, depending on the dump option in use when your program runs. For details about the dump options, refer to z/VSE System Control Statements.
- The partition in which your program runs.
- All general registers.

The generated dump is directed to the dump sublibrary for the partition if the job control option SYSDUMP is active. If the NOSYSDUMP option is active, the generated dump is printed on your SYSLST printer.

The job step (your program) is canceled if the macro is issued by the program's main (or only) task. If the macro is issued by a subtask, only this subtask is detached.

The JDUMP Macro

The functions of this macro are the same as those of the DUMP macro. However, the entire job is canceled if the program's main (or only) task issues the macro. If the macro is issued by a subtask, only this subtask is detached.

The PDUMP Macro

This macro provides, on SYSLST, a hexadecimal dump of:

- The general registers.
- The storage area between the addresses specified as two operands.

After having finished processing the macro, the system returns control to your program at the next instruction after the macro. Therefore, a PDUMP macro may be issued several times in a program to get dumps of selected storage areas at different stages of program execution.

The SDUMP and SDUMPX Macros

The SDUMP and SDUMPX macros provide support for data spaces and 31-bit addressing. The High Level Assembler is required to compile the macros.

The SDUMP and SDUMPX macros provide a fast dump of virtual storage which contains user data and system data. The output is written either into a dump library or onto SYSLST. The dump includes address

ranges in the primary address space as well as storage ranges in data spaces to which addressability via an ALET or via an STOKEN exists.

If the program is running in primary ASC (address space control) mode, either SDUMP or SDUMPX can be used. Otherwise, when the program runs in access register (AR) mode, the SDUMPX macro must be used. SDUMPX provides all of the functions of SDUMP, but generates code and addresses that are appropriate for AR mode.

If you are in access register (AR) mode, issue the SYSSTATE ASCENV=AR macro before you issue the SDUMPX macro to tell SDUMPX to generate code appropriate for access register mode.

The SDUMP macro cannot dump data space storage. To dump data space storage, SDUMPX is to be used instead by including either the LISTD or SUMLSTL operand.

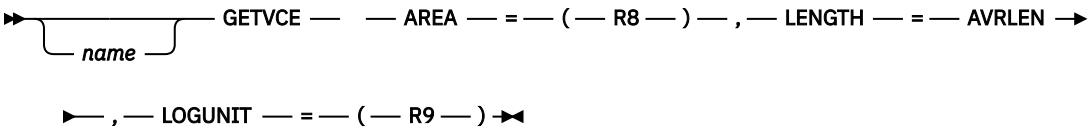
Note: A certain number of characters on every line of the printed dump may get lost if (1) your dump is directed to SYSLST assigned to a 3211 printer and (2) indexing is being used. To avoid this, load an FCB (forms-control buffer) image without an indexing byte before you issue the dump request macro. You can use the LFCB macro for this purpose.

Requesting Volume and Device Characteristics

You can request the system to return device and volume characteristics of a specific device by using the GETVCE macro. The macro also returns information about the track capacity and track balance of the specified device.

Retrieving Volume and Device Characteristics

The example shows how to retrieve device information.



AREA=(8)
Points to an area where the volume characteristics are to be stored.

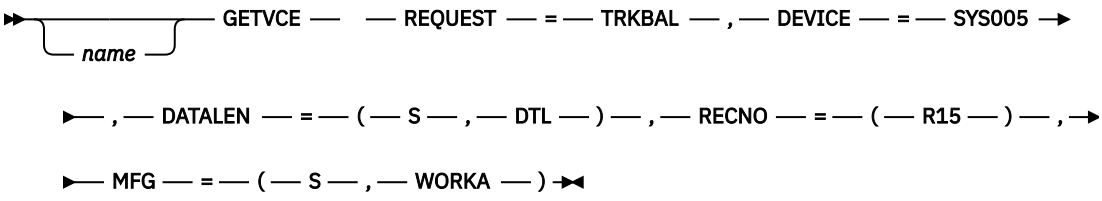
LENGTH=AVRLEN
Specifies the length of the data to be placed into the AREA field. In this example the length is specified by field AVRLEN generated by the macro AVRLIST. This is also the default if the LENGTH operand is omitted.

LOGUNIT=(9)
Points to a halfword that contains the logical unit name of the device.

The area pointed to by AREA is cleared in the specified length. The output moved into the specified area is described by a DSECT generated by the AVRLIST macro.

Obtaining the Track Balance of a Device

The example queries how many data bytes are left on the track after the specified record is written.



REQUEST=TRKBAL
TRKBAL indicates that the track balance is to be retrieved, that is the number of remaining data bytes on a track of the specified DASD.

DEVICE=SYS005
Specifies the logical unit number of the device.

DATALEN=(S,DTL)
Points to a two-byte field containing the length of one fixed-length data record. This field is processed as an unsigned binary value.

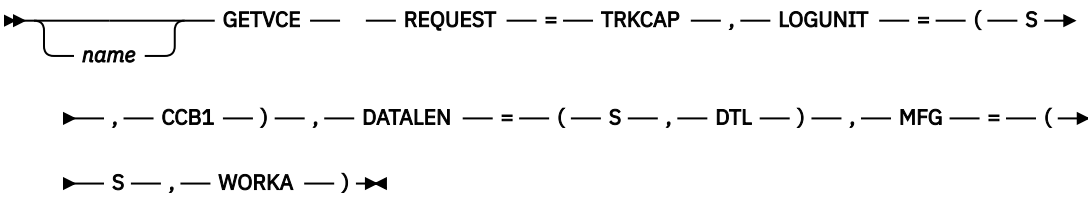
RECNO=(R15)
Points to a one-byte field containing the number of the record about to be written.

MFG=(S,WORKA)
Specifies the address of a dynamic storage area that is to be used if the program is to be reenterable.

On return, register 0 contains the updated track balance if the new record fits. If the (whole) new record would not fit, register 0 is set to zero.

Obtaining the Track Capacity of a Device

The example queries how many records of a given size fit onto the remainder of the track.



REQUEST=TRKCAP
Indicates that the capacity of the track is to be retrieved, that is the number of whole records that fit into the given or calculated track balance (remainder of the track) of this DASD.

LOGUNIT=(S,CCB1)
Points to a halfword containing the logical unit number specified in CCB1.

DATALEN=(S,DTL)
Points to a two-byte field containing the length of one fixed-length data record. This field is processed as an unsigned binary value.

MFG=(S,WORKA)
Specifies the address of a dynamic storage area that is to be used if the program is to be reenterable.

On return, register 0 contains the updated track capacity, that is the number of whole records that will fit onto the remainder of the track. Since RECNO is not specified in this example, the default is zero, which means that the number of whole records fitting on a complete track will be returned in register 0.

Requesting System Information

Macros are available to inquire about certain system information:

- SUBSID macro

  You can use the macro to inquire about the supervisor that controls your system. The macro retrieves a string of data that your program can interpret by using the mapping DSECT generated by the MAPSSID macro. You can, for example, check the mode in which your current supervisor has been generated, or whether it includes DASD sharing support.

- EXTRACT macro

  This macro, together with the DSECT generating macro MAPEXTR (or MAPBDY and MAPBDYVR) allows your program to get information about the boundaries of the partition in which the program runs.

Together with the mapping macro MAPEXTR MODE=SYSP (or MAPSYSP), the EXTRACT macro allows you to retrieve information about the system layout, such as begin and end address of supervisor area, SVA or GETVIS area.

In addition, your program can use the EXTRACT macro to inquire about the assignment status of a logical unit name.

Writing and Deleting Messages (WTO, WTOR, and DOM Macros)

The WTO and WTOR macros allow you to write a message to a display device, to a program that receives WTO and WTOR messages, or just to the hardcopy file. Besides writing a message, WTOR allows you to request a reply from the operator who receives the message. The DOM macro allows you to delete a message that is already written to the operator.

Routing the Message

You can route a WTO or WTOR message to a console by specifying one or more of the following parameters:

- ROUTCDE to route messages by routing code
- CONSID to route messages by console ID
- CONSNAME to route messages by console name

The ROUTCDE parameter allows you to specify the routing code or codes for a WTO and WTOR message. The routing codes determine which console or consoles receive the message. Each code represents a predetermined subset of the consoles that are attached to the system, and that are capable of displaying the message. WTO and WTOR allow routing codes from 1 to 128. Consoles provided by z/VSE are designed to receive all routing codes for master consoles, or none for user consoles.

You can also use either the CONSID or CONSNAME parameter to route messages. These mutually exclusive parameters let you specify the ID or the name of the console that is to receive the message. When you issue a WTO or WTOR macro that uses either the CONSID or CONSNAME parameter with the ROUTCDE parameter, the message or messages will go to all of the consoles specified by both parameters.

The MCSFLAG parameter is used to specify whether the message is a command response or for the hardcopy file only. Specifying MCSFLAG=BUSYEXIT will terminate the WTO if no console buffers are available. Control is returned to the issuer with a return code of X'20'. If you do not specify BUSYEXIT, WTO processing may place the WTO invocation in a wait state until WTO buffers are again available.

Altering Message Text

The TEXT parameter on the WTO macro enables you to alter repeatedly the same message or numerous messages. You can alter the message or messages in one of two ways:

- If you issued 3 different messages, all with identical parameters other than TEXT, you created a list form of the macro, moved the text into the list form, then execute the macro. Using the TEXT parameter you can use the standard form of the macro, and specify the address of the message text. By reducing the number of list and execute forms of the WTO macro in your code, you reduce the storage requirements for your program.
- If you need to modify a parameter in message text, using the TEXT parameter enables you to modify the parameter in the storage that you define in your program to contain the message text, rather than modify the WTO parameter list.

Using the TEXT parameter on WTO can reduce your program's storage requirements because of fewer lines of code or fewer list forms of the WTO macro.

Writing a Multiple-Line Message

To write a multiple-line message to one or more operator consoles, either issue WTO with all lines of text, or issue each line of text separately using the CONNECT parameter on the WTO macro.

The CONNECT parameter connects a subsequent message to a previous message. For example, if your program develops a large, multiple-line message of unknown length, it can issue several WTOs or the different parts of the message at different times. The CONNECT parameter forces all these WTOs to use the same message ID. These messages are combined into blocks of up to 12 lines, and delivered as a single message. CONNECT is mutually exclusive with CONSID, and it is not available with WTOR.

You can create with one WTO macro request a message that consists of up to 10 lines. For more than 10 lines, issue more than one WTO macro. The additional lines appear at the end of the message and continue until you specify an "END" line by specifying "DE" or "E" as the line type for the last line of data.

After processing the first request, the system places a message identifier in register 1. For each additional request, you must pass this identifier to the subsequent lines through the CONNECT parameter of WTO.

Deleting Messages Already Written

The DOM macro deletes the messages that were created using the WTO or WTOR macros. Depending on the timing of a DOM macro relative to the WTO or WTOR, the message may or may not have already appeared on the operator's console.

- When a message already exists on the operator screen it is highlighted to alert the operator that some action must be taken. When the operator responds to a message, highlighting is removed to indicate that a response was already given. When DOM deletes a message, it does not actually erase the message. It only resets the highlighting attribute, displaying it like a non-action message.
- If the message is not yet on the screen, DOM resets the highlighting before it appears. The DOM processing does not affect the logging action. That is, if the message is supposed to be logged, it will be, regardless of when or if a DOM is issued. The message is logged in the format of a message that is waiting for operator action.

The program that generates an action message is responsible for deleting that message.

To delete a message, identify the message by using the MSG parameter on the DOM macro.

When you issued WTO or WTOR to write the message, the system returned a message ID in general purpose register 1. Use the ID as input on the MSG parameter.

WTO, WTOR, DOM Usage Examples

```
         WTO   TEXT=MSG,                                          x
               ROUTCDE=(2,11)          * Route to master and user
         ...
         ST    R1,MSGID                * Save message ID
         ...
MSG      DC    AL2(L'MSGT)             * Response heading
MSGT     DC    C'...'                  * Up to 125 characters
MSGID    DS    F
```

Figure 56. Example of Single-Line WTO

```
         LA   R2,LINE
         WTO  TEXT=((TITL,C),((R2),D),(LAST,DE))                       X
              ROUTCDE=2,DESC=2     * Route to master, system status
         ...
TITL     DC   AL2(L'TITLT)         * Message title
TITLT    DC   C'...'               * Up to 34 characters
         ...
LINE     DC   AL2(L'LINET)         * Message data line
LINET    DC   C'...'               * Up to 70 characters
         ...
LAST     DC   AL2(L'LASTT)         * Message data line
LASTT    DC   C'...'               * Up to 70 characters
```

Figure 57. Example of Multiple-Line WTO

```
         SPLEVEL SET=3             * Request a VSE/ESA 1.3 expansion
         WTO   '...',ROUTCDE=11
         ...
         (no message ID is returned)
```

Figure 58. Example of a Backward-Compatible WTO

```
         (receive command from INCONS with INCART)
         ...
         WTO   TEXT=((HEAD,L)),                                        X
               CONSID=INCONS,CART=INCART,                              X
               MCSFLAG=RESP        * Indicate command response
         ...
         ST    R1,MSGID            * Save message ID
         ...
         WTO   TEXT=((LINE,D)),                                        X
               CONNECT=MSGID       * Connect to previous message
         ...
         WTO   TEXT=(,E),
               CONNECT=MSGID       * Ending line
         ...
MSGID    DS    F
HEAD     DC    AL2(L'HEADT)        * Response heading
HEADT    DC    C'...'              * Up to 70 characters
LINE     DC    AL2(L'LINET)
LINET    DC    C'...'              * Data line, up to 70 characters
INCONS   DS    F                   * Target console
INCART   DS    CL8                 * Command/response correlation token
```

Figure 59. Example of WTO for Command Responses

```
         WTOR  TEXT=(PROMPT,REPLY,L'REPLY,ECB),                        X
               RPLYISUR=INCONS     * Identify the reply origin
         ...
PROMPT   DC    AL2(L'PROMPTT)      * Response heading
PROMPTT  DC    C'...'              * Up to 122 characters
REPLY    DC    CL120' '            * May receive up to 119 characters
ECB      DC    F'0'
INCONS   DS    F                   * ID of replying console
INNAME   DS    CL8                 * Name of replying console
```

Figure 60. Example of WTOR

```
         WTO   TEXT=AMSG,                                              X
               DESC=2              * Immediate action message
         ST    R1,MSGID            * Save message ID
         ...
         (wait for action to take place)
         ...
         DOM   MSG=MSGID           * Delete message
         ...
AMSG     DC    AL2(L'AMSGT)
AMSGT    DC    C'... '             * Up to 125 characters
MSGID    DS    F
```

Figure 61. Example of Application Control of Message Deletion (DOM)

Example of an LBSERV MOUNT Request

This is an example of how to mount and release a tape via the LBSERV macro:

```
        SPACE 2
         IJJLBSER DSECT=YES
TESTCASE START X'78'
         BALR  R11,0                   SET UP
         USING *,R11                   ADDRESSABILITY
TLIB1    LA    R4,MYCUU
         LA    R5,VOLSER
         LA    R6,MYECB
         LA    R7,LIBNAME
         LA    R8,SERVL
         LA    R9,WRITEFL
         LA    R10,TCATEG
         LA    R13,SAVEAREA
         USING IJJLBSER,R8
         MVI   IJJLTLEN+1,IJJLTLN
*****************************************************************
*   LBSERV MOUNT
*****************************************************************
         LBSERV FUNC=MOUNT,VOLSER=(R5),ECB=(R6),LIBNAME=(R7),          X
               SERVL=(R8),WRITE=(R9),CUU=(R4),TGTCAT=(R10)
*
         LTR   R15,R15
         BNZ   EOJ

         LA    R1,MYECB
         WAIT  (1)
*****************************************************************
*   LBSERV RECEIVE
*****************************************************************
         LBSERV FUNC=RECEIVE,SERVL=(R8)
         CLC   IJJLTRET,RETCODE
         BNE   EOJ
*****************************************************************
* WRITE TO TAPE
*****************************************************************
         OPEN  FILE
         LA    R9,10
LOOP1    AP    NUMBER,CONST
         UNPK  RECNUMB,NUMBER
         OI    RECNUMB+9,X'F0'
         PUT   FILE,RECORD
         BCT   R9,LOOP1
         CLOSE FILE
```

```
*****************************************************************
*   LBSERV RELEASE
*****************************************************************
         LBSERV FUNC=RELEASE,ECB=MYECB,SERVL=(R8),CUU=MYCUU
*
*

         LTR   R15,R15
         BNZ   EOJ

*
         LA    R1,MYECB
         WAIT  (1)
*****************************************************************
*   LBSERV RECEIVE
*****************************************************************
*   REGISTER NOTATION
         LBSERV FUNC=RECEIVE,SERVL=(R8)
*
         CLC   IJJLTRET,RETCODE
         BNE   EOJ
*
EOJ      DS    0H
         EOJ
IOAREA1  DC    800CL1' '
RECORD   DC    C' RECORD NUMBER'
RECNUMB  DC    C'0000000000'
CONST    DC    X'00001C'
NUMBER   DC    X'00000C'
FILE     DTFMT BLKSIZE=800,                                          *
```

```
                RECSIZE=80,                                            *
                RECFORM=FIXBLK,                                        *
                TYPEFLE=OUTPUT,                                        *
                DEVADDR=SYS005,                                        *
                WORKA=YES,                                             *
                IOAREA1=IOAREA1,                                       *
                HDRINFO=YES,                                           *
                FILABL=STD,                                            *
                EOFADDR=EOJ
*
```

```
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
*
*   WORK AREAS AND CONSTANTS:
*
         DS    0D
*
SERVL    DC    (IJJLTLN)X'00'
MYECB    DC    F'00'
MYCUU    DC    CL4'0061'
WRITEFL  DC    CL1'W'
VOLSER   DC    CL6'099275'
RETCODE  DC    CL4'0000'

LIBNAME  DC    CL8'MYLIB001'
TCATEG   DC    CL10'SCRATCH01 '

SAVEAREA DC    18F'0'
         END
```

Library Access for Application Programs

The Library Access Service enables application programs to access library objects in a way similar to other data access methods.

The access service uses two assembler macros, LIBRM and LIBRDCB, as described in z/VSE System Macros Reference.

Storage Requirements

The access service dynamically allocates partition GETVIS storage for internal buffers, control blocks, and data areas. Some of these areas are fixed, others are variable in size.

The size of the variable part depends on the

- Specific LIBRM request
- Track capacity of the DASD where the library resides
- Size of the library member accessed
- Size of the DIRINF area specified in the LIBRM request

Therefore, for good performance with a minimum of SIO, things like DASD capacity and member size must be considered.

The size of the DIRINF area automatically enlarges internally used tables, even if these tables are not filled by directory information (for example, generic requests).

The fixed part of GETVIS storage is about 24K, whereas the dynamic part can vary from 2K up to 100K. A normal application should consider a GETVIS area in the range from 30K up to 60K to be used by the access service.

Record I/O

Record I/O offers a convenient way to access and retrieve library member data in pieces, either as records or substrings. The access service is designed to process single records or bytes with GET/PUT requests.

The member to be worked on can be specified in two different ways:

- It can be fully qualified as 'lib.sublib.member.type' with the requested service, for example, OPEN member.
- It can be partially qualified as 'member.type'. The 'lib.sublib' specification is then taken from a sequence (chain) of sublibraries addressed by a name (chain ID) and defined either by the LIBRM LIBDEF macro or by the external LIBDEF job control statement.

If the sublibrary is explicitly specified (by 'lib.sublib'), the function is performed as for a chain of sublibraries with that sublibrary being the only element in the chain.

Librarian Control Block

Before working on a library object (library, sublibrary, member or chain), its characteristics and processing requirements have to be specified in the Librarian Data Control Block (LDCB). An LDCB is required for each LIBRM request and is created in your processing program by a LIBRDCB macro instruction.

Information placed in the LDCB is either taken from the LIBRDCB or LIBRM macro instructions, or retrieved by a previously installed sublibrary chain or existing library object (sublibrary or member). If more than one source specifies information for a particular field of the LDCB, only one source is used.

Accessing or Updating Librarian Member Data

To get access to member data, a member must first be opened. There are three types of OPEN: OPEN for input (INPUT), OPEN for output (OUTPUT), and OPEN for input/output (INOUT).

- OPEN(INPUT) provides read-only access to an existing member. After OPEN, the following requests are possible: GET, NOTE, and POINT. If a sublibrary search chain is specified, the member is opened in the sublibrary where it is found first.
- OPEN(OUTPUT) provides write-only access to a new member (sequential file). The replacement of an existing member with the same member and type name is controlled by the REPLACE/NOREPLACE option. After OPEN, only PUT and NOTE requests for member data are accepted. An OPEN(OUTPUT) works only on the first sublibrary of a chain.
- OPEN(INOUT) provides read/write access to a member. If the member does not exist, a new member is created. If the member exists, an output copy of the member is generated that is initially empty. PUT requests to the member write the data to this copy of the member, and the copy replaces the original member at CLOSE time. If a sublibrary chain is specified, the member is opened and replaced in the sublibrary where it is found first. If the member does not exist, it is created in the first sublibrary of the chain.

  To modify the member, combine the sequential copying of the existing data (by LIBRM GET and LIBRM PUT) with appropriate PUT calls to alter the data as necessary. For example::

  – To simply overlay records or bytes, modify the data in the buffer after calling GET and before calling PUT.
  – To insert data within the member, copy using GET/PUT up to the point of insertion, PUT the new data, and continue the copy process.
  – To append data to the member, copy the entire member using GET/PUT and then issue the PUT call for the new data. It is not possible to append data to a member without first copying the existing member to the output copy of the member.

- To insert records or bytes at the beginning of the member, PUT this data before copying the existing data.
- To delete data, omit the corresponding PUT call.

With the LIBRM NOTE macro it is possible to extract the current (read or write) position within a member and use this information later for repositioning with the LIBRM POINT macro. Like GET, POINT works on input members only (opened for INPUT and INOUT). The information returned by NOTE contains both the Logical Relative Byte Address (LRBA) and the Physical Relative Byte Address (PRBA) within the member, so that locating can be done without re-reading the first part of the member. The NOTE information must be passed to the POINT function without change.

If the member is opened for INOUT, POINT can be used only to position within the original member, not within the output copy of the member. For example, you cannot simply position to the end of the member using POINT and then append the desired data. The existing data must first be copied as described above.

The NOTE information can be handled in two different ways:

1. It is managed by the system in a last-in first-out manner by a stack with a nesting depth of 20 entries.
2. The user has control over the NOTE information.

It is possible to do NOTE/POINT processing not only within a member, but also over different members in a sublibrary chain. Repositioning over member boundaries is only supported for members opened for INPUT in a specific chain in the following sequence:

- OPEN INPUT member1 CHAINID=xx
- GET records of member1
- NOTE the current read position of member1 (only NOTECTL=NO possible)
- OPEN member2 (the chain of the first OPEN will be considered)
- GET records of member2
- POINT back to member1
- GET next record of member1
- CLOSE complete processing

With the LIBRM CLOSE macro you can stop all access to a library member and cause all resources (GETVIS space, locks, for example) to be given up. For a newly created member (OPEN for OUTPUT), the library directory entry is cataloged or updated, unless the update of the directory is prevented (with LIBRM CLOSE COMMIT=NO).

Library Access Functions

The library access service supports the following functions:

- LIBDEF chainid,chain
  - where a sequence **chain** of sublibraries with a given name **chainid** is established which can be searched for individual sublibrary members.
- LIBDROP chainid
  - where a chain of sublibraries with the given name **chainid** is dropped.
- STATE member or STATE member,chainid
  - which checks whether the specified **member** exists in a certain sublibrary or chain of sublibraries. If it exists, the access service returns member attributes like number of records, record format, and record length. If the member is searched for in a chain of sublibraries, the system also returns the name of the first library and sublibrary in which the member resides.
- STATE sublib

  - which checks whether the sublibrary **sublib** exists in a certain library. If it exists, the access service returns sublibrary attributes like number of members, used space, and space re-usage options.

- STATE lib
  - which checks whether the library **lib** exists. If it exists, the access service returns library attributes like number of sublibraries, library size, used and free space.
- STATE chain
  - which checks whether a sublibrary **chain** identified with chainid exists. If it exists, the sublibrary names are returned to the caller.
- LOCK member
  - which locks a member for any write or update access.
- UNLOCK member
  - which frees a member from a previously given LOCK request.
- DELETE member or DELETE member,chainid
  - which deletes member **member** (within sublibrary chain **chainid**).
- RENAME oldname,newname
  - which renames a member with **oldname** to **newname**.
- OPEN member or OPEN member,chainid
  - which opens a **member** (within the chain **chainid**) for reading/writing of records.
- GET bufferaddr,bufferlen
  - which reads one or more records or bytes from a member into the buffer described by **bufferaddr** and **bufferlen**.
- PUT ldcbname,bufferaddr,bufferlen
  - which writes one or more records or bytes to a member from the buffer described by **bufferaddr** and **bufferlen**.
- NOTE
  - which notes the position within a member to be able to point at it later.
- POINT
  - which points to a position within a member noted earlier.
- CLOSE
  - which closes a member and denies further (read/write) access to it. For a newly created member you can specify whether you want to commit or decommit its cataloging into the library directory.

Return Code Conventions

Each library access service macro passes back return information to inform the invoking program about the completion of the requested function. This return information consists of two parts,

- A return code (severity code), passed in register 15, which gives a global statement about the success or failure of the requested service, and
- A reason code, returned in register 0, which informs the caller in detail about any failing condition.

The codes are used uniformly throughout the library access service macros.

The meaning of the return codes is as follows:

- RC=0 indicates successful completion. The service worked as requested.
- RC=4 indicates that either the requested function was performed, but an exceptional condition exists, or that the function was not performed because the requested result already exists.
- RC=8 means that some functions are not or only partially executed, but processing was continued.
- RC=12 is returned if the requested service could not be performed at all, because the addressed library resource was not available.
- RC=16 indicates that there is an externally controllable condition (for example, lack of resources such as storage space) which resulted in the failure. This return code is accompanied by a Librarian message.
- RC=20 indicates an error condition as a result of internal Librarian processing. This result is accompanied by a Librarian message.
- RC=32 is given for unauthorized access to a library object. This return code is given together with message L163I.

If a message is generated by the Librarian as a result of return code 16 or higher, it is passed back in a buffer within the LDCB (label IALCMSG); it is not written onto SYSLST/SYSLOG. The reason codes are defined individually for each service.

Apart from the return code in register 15 and the reason code in register 0, the library access service provides exits to handle the different levels of error situations. An exit can be specified for an 'Object Not Found', for an 'End of Member', and for an 'Unexpected Error' condition, respectively.

An additional processing option exists for unexpected errors (that is, return code 16 and higher) to decide whether processing should be canceled, transferred to an exit, or returned to the user invocation.

See “Librarian Exits” on page 144 for a detailed description of the exits.

Record Formats

The supported record formats are: FIXED and STRING.

FIXED denotes logical records of fixed length. The size of fixed-length records is the same for all records in a member. An access request refers to the record number and the number of records to be processed. FIXED records are 80 bytes long.

STRING means a format where the whole member is considered as a byte string. An access request refers to a substring thereof, described by the LRBA (logical relative byte address) of the first byte and the length of the substring. This format allows to interpret the data in any other way by a program.

Processing Sequence of Sublibrary Chains

The following search sequence applies when sublibrary chains are processed by OPEN, STATE(MEMBER), DELETE, and RENAME requests with the CHAINID=chainid operand:

1. Task-related chain with the specified 'chainid'.
2. Job-related search chain (only for CHAINID=PHASE, OBJ, PROC, and SOURCE).
3. Partition-related search chain (only for CHAINID=PHASE, OBJ, PROC, and SOURCE).
4. System-related sublibrary IJSYSRS.SYSLIB (only for CHAINID=PHASE, OBJ, PROC, and SOURCE).

The described chains are concatenated when performing a search for a member. For OPEN(OUTPUT) only the first sublibrary will be selected.

Note: The chain specified with the job control statement

```
// LIBDEF PHASE,SEARCH=(...,SDL,..),TEMP
```

cannot be processed by the library access service (return code 16 with the corresponding reason code is returned).

Register Usage

Register
Usage

0
Reason code

1
Address of LDCB

2-12
Not used

13
Address of caller-provided save area (72 bytes)

14
Return address to macro invocation

15
Return code

For the register notation used with the librarian exits refer to “Librarian Exits” on page 144.

Librarian Exits

ERRAD: Librarian Error Exit

With the ERRAD=label operand in the LIBRDCB macro you can specify a label to which the Librarian will branch if an unexpected processing error (return code > 12) occurs. All necessary clean-up processing is automatically done before the exit gets control. The ERRAD exit is not a subroutine, but rather a continuation of the requester routine.

When the caller receives control to the specified label, the register values are as follows:

Register
Value

0
Librarian feedback code

1
Address of failing LDCB

2-12
As before the macro invocation

13
Address of the user-provided save area (72 bytes)

14
Address of ERRAD label (branch address)

15
Return code

Following are the committed values in the LDCB, which can be used for error analysis:

IALCCMD
Failing LDCB command (macro request)

IALCLIB
Address of requested library name (if specified), otherwise 0

IALCSLIB
Address of requested sublibrary name (if specified), otherwise 0

IALCMEMB
Address of requested member name (if specified), otherwise 0

IALCTYPE
Address of requested member type (if specified), otherwise 0

IALCNMBR
Address of requested new member name (if specified), otherwise 0

IALCNTYP
Address of requested new member type (if specified), otherwise 0

IALCCHID
Address of requested chainid (if specified), otherwise 0

IALCLKID
Address of requested lockid (if specified), otherwise 0.

IALCRETC
Return code

IALCFDBC
Feedback code

IALCMSGA
Address of the librarian message area (120 bytes), always filled with the failing Lxxx message.

IALCNRET
Address of the instruction after the failing request (normal return address).

EODAD: Librarian 'End-of-Member Data' Exit

The EODAD=label operand of the LIBRDCB macro defines a label to which the Librarian will branch if there is no more member data available for any subsequent LIBRM GET requests. The member remains opened as for a normal return. A LIBRM CLOSE can be issued to finish processing.

Like ERRAD, the EODAD exit is also not a subroutine, but rather a continuation of the requester routine.

When the user receives control to the specified label, the register values are as follows:

Register
Value

0
Reason code (which is zero)

1
Address of active LDCB

2-12
As before the macro invocation

13
Address of the user-provided save area (72 bytes)

14
Address of EODAD label (branch address)

15
Return code (which is 8)

NOTFND: Librarian 'Not Found' Exit

The NOTFND exit will be taken if a specified library object does not exit. A library object is also treated as not existing if one of the related 'higher' objects does not exist or is not accessible (return code 12). For example, the NOTFND condition is also true for a member request if the specified sublibrary or library does not exist.

For the DELETE MEMBER request the NOTFND exit will not be taken at all.

When the caller receives control to the specified label, the register values are as follows:

Register
Value

0
Reason code

1
Address of active LDCB

2-12
As before the macro invocation

13
Address of the user-provided save area (72 bytes)

14
Address of NOTFND label (branch address)

15
Return code

Processing of Executable Programs (Phases)

The Library Access Service does not support the modification or creation of members of type PHASE (executable programs). It is recommended to use the standard utilities, such as the MSHP CORRECT function or the Linkage Editor to do this.

Library Access Request Sequence

The table shows the allowed sequence of access service requests belonging to one LDCB. It is possible to have more than one LDCB active in a user program.

n'th request	n+1'th request: OPEN INPUT	OPEN OUTPUT	OPEN INOUT	GET	PUT	NOTE	POINT	CLOSE	any other
OPEN-INPUT				x		x	x	x	
OPEN-OUTPUT					x	x		x	
OPEN-INOUT				x	x	x	x	x	
GET	(5)			x	(1)	x	(3)	x	
PUT				(1)	x	x	(1)	x	
NOTE	(2)			(3)	(4)	x	(3)	x	
POINT	(2)			x	(1)	x	x	x	
CLOSE	x	x	x					(6)	x
any other	x	x	x					(6)	x

x
There is no restrictions using this sequence.

(1)
Member must be opened for INOUT.

(2)
Only possible for NOTECTL=NO and member opened for INPUT.

(3)
Member must be opened for INPUT or INOUT.

(4)
Member must be opened for OUTPUT or INOUT.

(5)

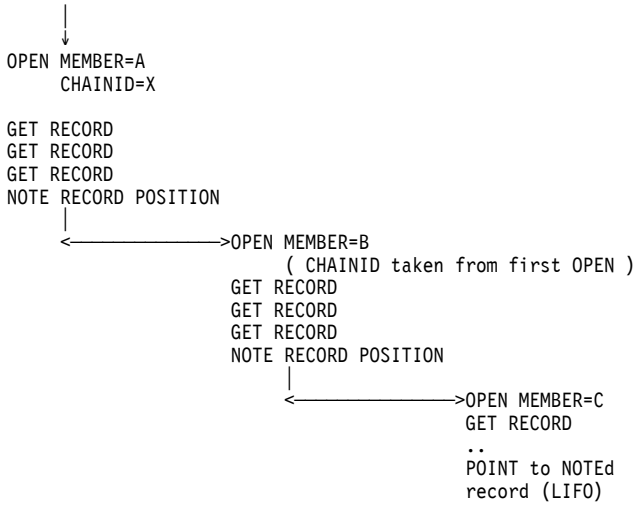
A new OPEN after the GET is only possible after a previous NOTE with NOTECTL=NO (stacked source inclusion).

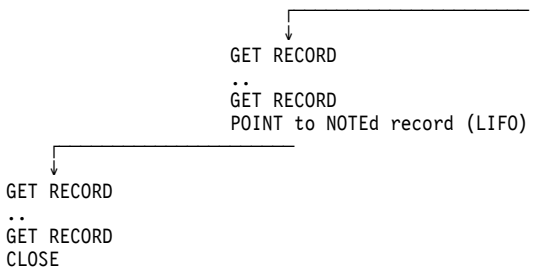
(6)
This CLOSE results in a NOOP.

Figure 62. Library Access Request Sequence

Example of an Access Service Request Sequence

The request sequence describes what a stacked read access might look like. The same LDCB is always used.





Cross-Partition Communication

The cross-partition communication service (XPCC) allows communication between two application programs (two different VSE tasks). It is invoked via the XPCC macro. For an example of the use of this macro in an application program requesting VSE/POWER services, see VSE/POWER Application Programming. The main XPCC functions are:

- An identify function (IDENT) allows application programs to make themselves known (to 'log on') to XPCC. XPCC recognizes the names of the applications and uses those names later to set up a communication path between the applications.
- Before data can actually be transmitted between two XPCC users, a communication path has to be established. The applications have to build this path via the CONNECT function. In order to have a complete link, **both** applications have to request the connection. After that they can start exchanging data via this path. The link is always a two-way-only communication path, that is, a data transmission request is always directed to only one other application.

  Before a program can issue a request, it must ensure that the preceding request (if any) is complete. For this purpose a WAIT capability is provided.
- Once the data transmission link is completed, the two applications can start exchanging data. XPCC will make sure that the sender of the data does not overwrite any user data in case data is sent faster than the receiver can process the data. Whenever a request is issued, return information about the other side of the communication path is provided.

  Sending and receiving data is done asynchronously with a WAIT and POST capability.
- Special commands are provided for clearing a connection from a data transmission request.
- Applications disconnect a communication path with the DISCONN or DISCPRG function and finally terminate their communication with the TERMINATE function. After that they are no longer known to XPCC.
- In case of normal or abnormal task termination, XPCC automatically disconnects any outstanding communication paths and, in addition, does a 'logoff' (TERMINATE) for the corresponding application.

All XPCC requests are associated with a program-defined control block, the XPCCB (cross-partition communication control block), which is set up with the XPCCB macro. The XPCCB is used to

- Define request options
- Set up pointers to buffer areas and ECBs
- Receive system return information.

The XPCCB may be generated statically at assembly time or it may be set up and/or modified at execution time. The individual fields of the XPCCB can be referenced via the mapping macro MAPXPCCB.

A communication path is always represented by a unique XPCCB. This specific XPCCB must be used for all requests given between CONNECT and DISCONNECT of this path. Control block identifier IJBXSTRT and control block length IJBXLEN are checked with each XPCC request.

After each request, the system supplies return information in register 15 and in field IJBXRETC of the XPCCB. Whenever an ECB is posted, status (reason code) information is provided in field IJBXREAS. You should test this information along with testing the posting of any ECB in the XPCCB. For the mnemonics

that you can use to test the return and reason codes supplied by the system, see the MAPXPCCB macro in z/VSE System Macros Reference.

Register 15 indicates whether an XPCC request was successful:

0 (X'00')
Request was started successfully.

4 (X'04')
Same as X'00', but additional return information is stored in IJBXRETC.

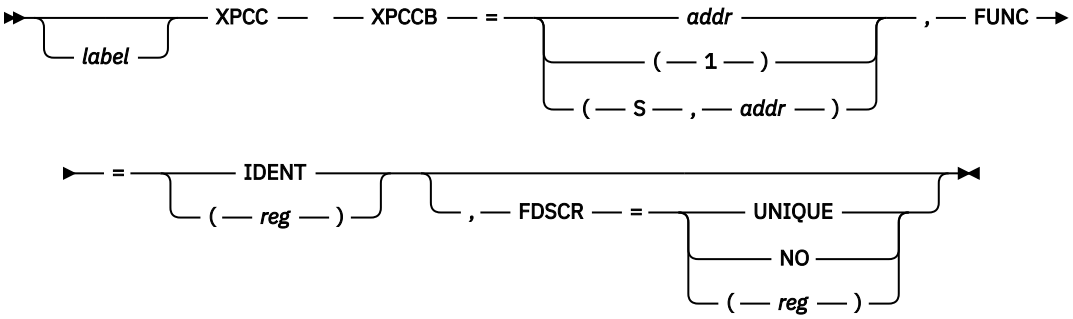
8 (X'08')
Request was rejected. IJBXRETC defines the reason for the failure.

12 (X'0C')
Request was rejected. The XPCCB address is invalid or length of XPCCB is invalid.

Identification of Communication User

To set up a connection between two applications, the applications must be known by the system.

Therefore, before requesting any XPCC services, the application has to identify itself to XPCC. This is done via the IDENTIFY function of the XPCC macro.



The fields used by the request in the XPCCB control block are:

Table 7. XPCCB control block input	
Name	**Description**
IJBXFCT	Function byte (IJBXID).
IJBXFDSC	Function descriptor byte.
IJBXAPPL	Application name of the requesting application. It can be up to 8 bytes long and must not contain blanks (X'40') or all binary zeros.

Table 8. XPCCB control block output	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXITID	TID (task ID) of IDENT requester.
IJBXIDK	Identify token provided by the system after the IDENT request is completed.
IJBXLGID	ID of partition using the application name that was requested unique.

XPCC associates the application with a unique identify token, which the VSE system returns in field IJBXIDK. All subsequent CONNECT requests issued by the application must copy this identify token into their own XPCCB.

One program can issue several IDENT requests with different application names. This means that the program is known under two (or more) different names, each representing a different application.

Application names starting with 'SYS' are reserved for programs written by IBM.

If the IDENT request is issued by the VSE maintask, also the subtasks in the partition can use the identify token. If the IDENT request is issued under control of a VSE subtask, only this subtask can use the identify token.

If a task terminates, the identify tokens owned by the task are invalidated.

Defining a Communication Path

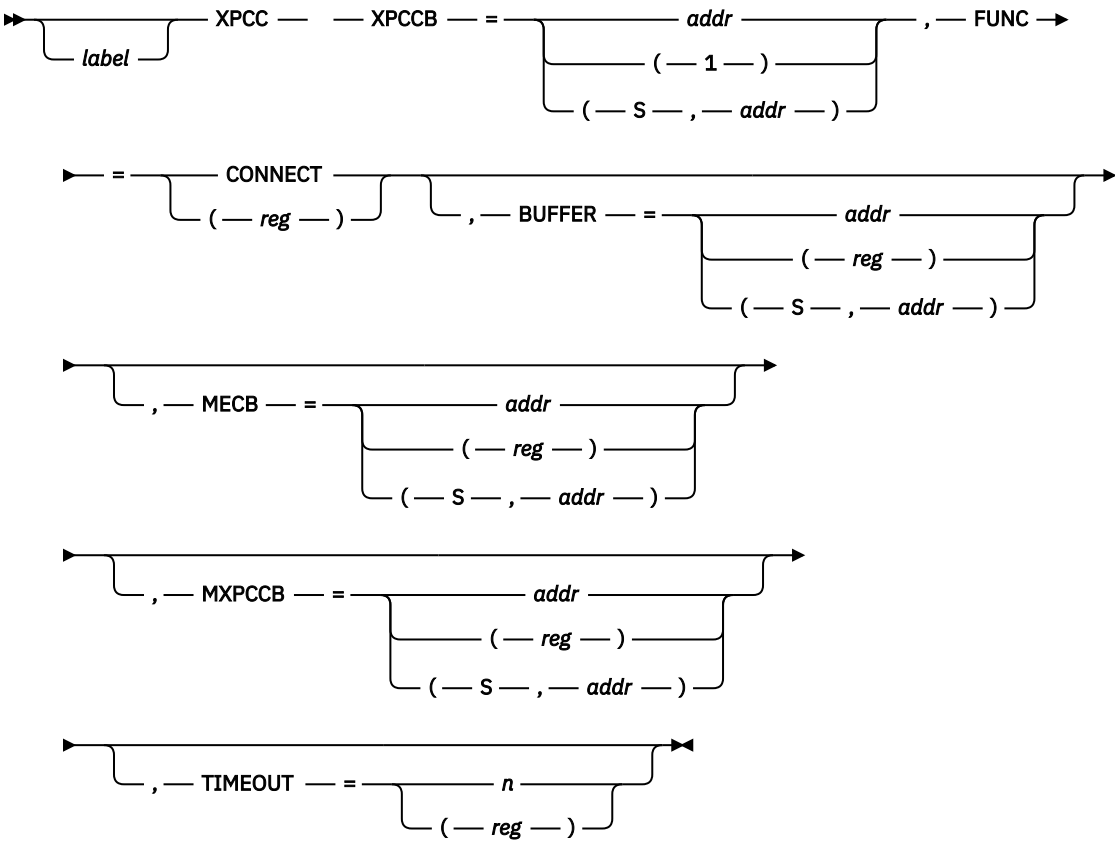
Before starting with the actual data transmission, you have to establish a unique communication path via the CONNECT request.

A communication path is always built between two applications. In order to get completed, the connection has to be requested by both applications.

Two types of connections are possible:

- A connection directed to a specific application.
- An 'open-ended' connection, to which any other application can link up.

Defining a Specific Connection



The CONNECT request uses the following fields in the XPCCB.

Table 9. CONNECT request XPCCB input fields	
Name	**Description**
IJBXFCT	Function byte (IJBXCON).

Table 9. CONNECT request XPCCB input fields (continued)	
Name	**Description**
IJBXFDSC	Set by TIMEOUT operand.
IJBXTOAP	Name of application (up to eight bytes) to which the connection is to be established. This is the name used by the other application at IDENTify time. If an 'open-ended' connection is to be set up, you have to initialize this field to binary 0. (TOAPPL=ANY if defined in the XPCCB macro.)
IJBXIDK	Identify token provided by the system after IDENT of requesting application. If MXPCCB is specified in the XPCC macro, the identify token of the specified XPCCB is copied into the current XPCCB before the CONNECT request is started.
IJBXSUSR	Eight bytes of user data to be moved into receiver's IJBXRUSR field after connection has been completed.
IJBXBUF	Consists of IJBXIND, IJBXAD31, and IJBXBLN. If only one buffer is specified (IJBXIND=B'1xxx xxxx'), the buffer is validated and saved (if found correct) for later use within SENDI protocol.
IJBXMECB	The ECB is validated and the address is saved, if found correct. Later on, the specified fullword is used as a main ECB, posted together with any XPCC ECB.
IJBXTIME	Specified WAIT time.

Table 10. CONNECT request XPCCB output fields	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXREAS	Reason code.
IJBXCTID	TID (task ID) of CONNECT requester.
IJBXPID	Communication path ID provided by the system when the connection is completed. It uniquely identifies the connection.
IJBXTOAP	Is updated in case of a CONNECT-ANY request with the application name of the partner.
IJBXCECB and IJBXSECB	Set to F'00'. They are posted whenever the connection is complete (data transmission can start).
IJBXRECB	Set to F'00'.
IJBXCNTL	Set to 4F'00'.
IJBXRUSR	User data from IJBXSUSR of the other side is filled in at connection completion time.
IJBXSILN	Length of partner's receive area for SENDI protocol.

CONNECT tries to establish a communication path to the application requested in IJBXTOAP.

If the other side has already issued the corresponding CONNECT, the connection is established right away and IJBXCECB and IJBXSECB are both posted.

If the other application has not yet issued CONNECT or is even not yet active, the request sets a return code in field IJBXRETC, but the connection request is accepted. IJBXCECB and IJBXSECB will be posted

and the task will be taken out of the wait state as soon as the other side issues the corresponding CONNECT, which completes the connection.

When IJBXBUF specifies only one buffer, the buffer is validated. Address and length are saved by XPCC and the length is stored into IJBXSILN of the partner's XPCCB (which must be a VERSION=2 XPCCB).

Connections are owned by the task that issued the CONNECT request. Any data transmission or DISCONNECT request can only be issued by the task that issued the CONNECT request. All requests issued between CONNECT and DISCONNECT must use the same XPCCB.

When the TIMEOUT parameter has been specified and the time interval has been exhausted, IJBXCECB, IJBXSECB, and IJBXRECB are posted and the task is taken out of the wait state.

Defining an Open-Ended Connection

When the application trying to establish a connection works as a server for the target application, it normally does not know the application name of the target. In such a case, the application would provide a connection to which any target application could connect - by specifying TOAPPL=ANY in the XPCCB macro.

For a CONNECT ANY request, the format of the XPCC request and the usage of the different XPCCB fields is the same as for a CONNECT-specific request.

Data Transmission

Once a connection has been established, the two applications can start exchanging data via this link.

The sender of the data builds up a list of data areas (and their length) and - via the SEND request - passes them to XPCC. At the other end of the connection, XPCC moves the corresponding control information (message length) into the IJBXCNTL field and posts the RECEIVE ECB (IJBXRECB). The other application would then realize that there is a pending SEND request. It would first inspect the control buffer for the length of the message (obtain optionally buffer space), and would then ask for the data transfer via the RECEIVE request, indicating where the data is to be stored.

The system transfers the data into the corresponding buffer space. At the same time it posts the SEND ECB associated with the SEND request at the sender's side (IJBXSECB), to indicate that the SEND request is successfully completed. At the receiver's side, the IJBXRECB is reset in order to be ready for the next SEND request.

As soon as a SEND or SENDR request is started on an available link, the connection is considered to be 'busy'. In case of a SEND, the connection is busy until the receiver issues the RECEIVE (this is also valid for a SEND with zero data length). In case of a SENDR, the connection is busy until the receiver accepts the data via RECEIVE and sends a reply back via REPLY. The receiver may also 'free' the connection by purging the data via the PURGE function.

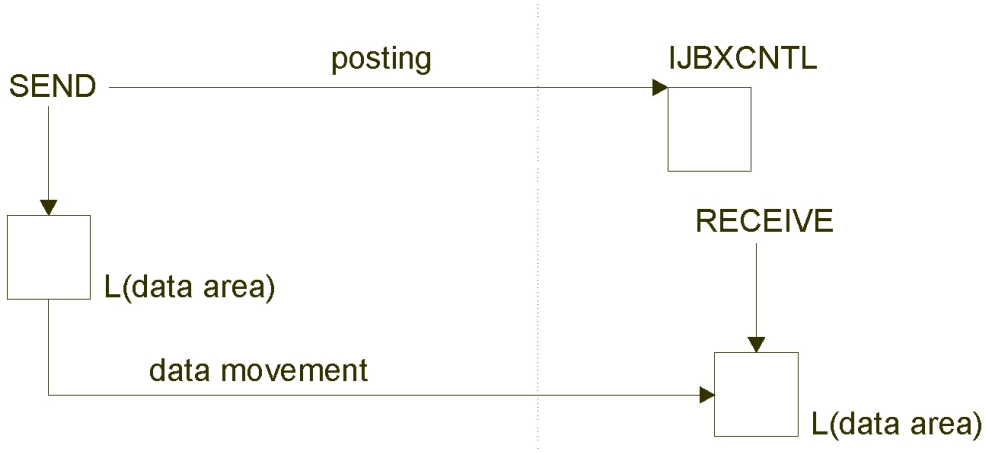
As long as a connection is busy, no new data transmission request can be started (it will be rejected with a return code).

The sender of the data may revoke the SEND request with a CLEAR request; however, the connection is still regarded as 'busy' until the receiver acknowledges the CLEAR by a RECEIVE, REPLY or PURGE.

Sending and Receiving Data

Two data transmission methods are possible:

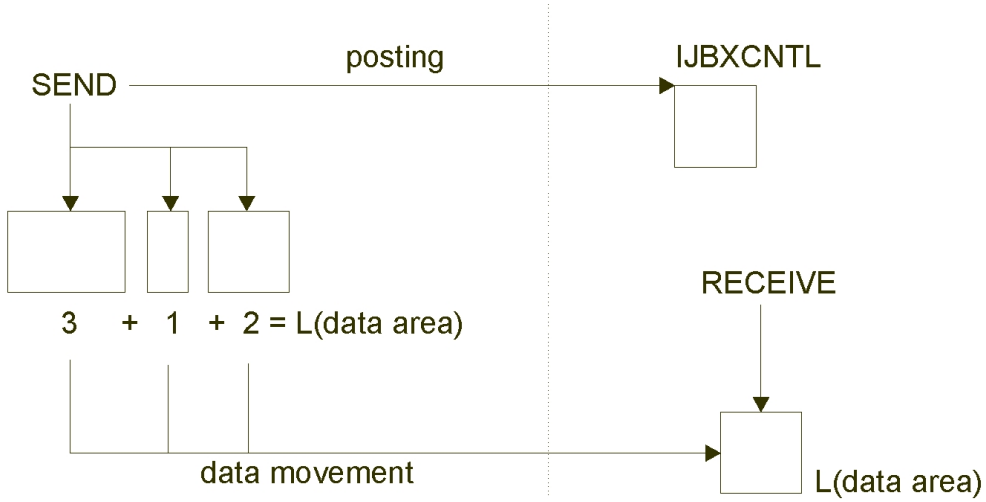
1. Transmission of data such that one data record on the sender's side is received as the same data record on the receiver's side:



The SEND request will post to the receiver's side the length of the data area to be sent (in the IJBXCNTL field). The receiver obtains the needed storage and issues RECEIVE.

2. Transmission of data such that the sender provides a list of data areas to be sent, and the receiver collects the concatenated data into one data area (the length being the sum of the sender's data area lengths):

   The concatenated data length is posted at SEND time to the receiver's side. The receiver dynamically obtains the needed storage and issues RECEIVE.



The programs may choose at SEND time whether to use method 1 or method 2.

Three protocols are available for data exchange:

1. A SEND - RECEIVE protocol:

   If this protocol is used, the sender is posted when the receiver has accepted the sent data via the RECEIVE request. At this time, the connection is available for the next data transmission request.

2. A SENDR - RECEIVE - REPLY protocol:

   When using this protocol, the sender requests a reply from the receiver. The sender is posted when the receiver sends a reply back upon receiving the data. XPCC transfers the reply data into a reply area, which has to be provided by the sender at SEND time (with the XPCCB REPAREA operand).

3. A SENDI - SENDI protocol:

   When using this protocol, SENDI must be issued alternately from both partners, where the first SENDI starts the communication. This protocol requires that both partners have defined buffers at CONNECT time. These buffers are used as receive buffers when SENDI is executed.

If the predefined receive buffer is too small, XPCC switches to the normal SEND-RECEIVE protocol. The receiver is notified by the switch in IJBXREAS. After the RECEIVE has been executed, return to the SENDI protocol is recommended. If an error situation forces one of the partners to break the SENDI sequence, switching to the SEND-RECEIVE protocol is possible.

The SENDI protocol with a data length of zero can be used as a simple WAIT/POST mechanism between tasks in different partitions.

Note: It is strongly recommended not to switch between the different protocols, because fields in the XPCCB might be overwritten by succeeding XPCC requests before the partner has a chance to inspect the original content.

Data Transmission without Reply Request

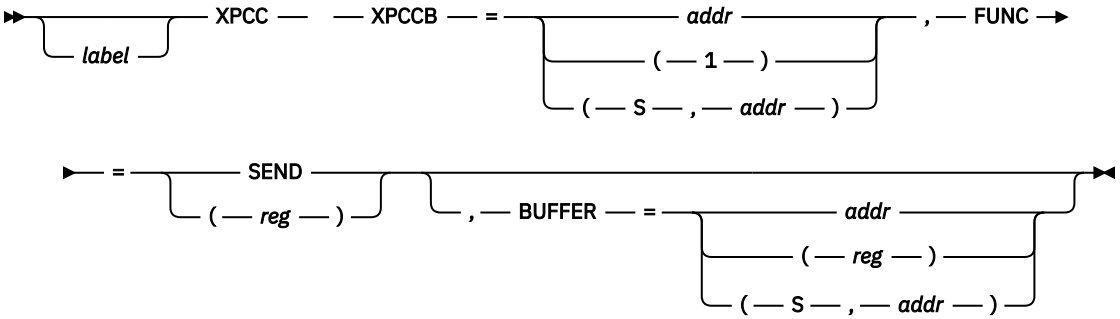


Table 11. SEND request - Input fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXCON).
IJBXDSC	Should be set to zero.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXBUF	Consists of IJBXIND, IJBXAD31, and IJBXBLN.
IJBXIND	B'1xxx xxxx' If only one data area is to be transmitted. B'0xxx xxxx' If a list of data area addresses is provided.
IJBXAD31	Address of data area to be transmitted or address of list of data area addresses.
IJBXBLN	Length of data area to be transmitted. Not used in case of data area list (IJBXIND=B'0').
IJBXSUSR	Eight bytes of user data to be moved into receiver's IJBXRUSR field.

Table 12. SEND request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXCECB	Reset wait bit.
IJBXRECB	Reset wait bit.
IJBXSECB	Reset wait bit (posted if other side issues RECEIVE or PURGE).

Table 13. SEND request - Output fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXREAS	Reason codes.
IJBXRECB	Posted by system.
IJBXRUSR	8 bytes of data from the sender's IJBXSUSR field.
IJBXSLN	Length of data to arrive or (in case of an address list) the sum of the length of the sender's data areas.
IJBXFLG	X'01' For a 'normal' SEND request.

Additional Information

If a list of data areas is provided, the IJBXADR field points to a list of 8-byte fields where each entry has the following format:

Table 14. IJBXADR 8-byte fields	
Bytes	**Description**
0	B'0' If this is not the last entry. B'1' If this is the last entry in the list.
0 - 3	Address of data area to be sent.
4 - 7	Length of data area to be sent.

The list can have up to 256 entries.

The addresses of the data areas to be transmitted and their lengths are passed to XPCC which will do an address validation. If the XPCC BUFFER operand is not used, the XPCCB information stored in the IJBXBUF field is used for data transmission. If the BUFFER operand is specified, it overwrites the information stored in the IJBXBUF field.

XPCC calculates the length of the data to be transmitted and moves this information into field IJBXSLN at the receiver's side. The data provided in field IJBXSUSR (send user data) of the sender will be moved into field IJBXRUSR (receive user data) at the receiver's side.

If the connection is 'busy', the SEND request is rejected. This might occur if the other side has not yet issued a RECEIVE for a previous SEND request, or the other side has issued SEND. In the first case the SEND request could be retried when the IJBXSECB associated with the previous SEND request is posted, in the second case a RECEIVE or PURGE request must first be issued.

The SEND function might be issued with the IJBXBLN field (length of data area) initialized to zero and IJBXIND initialized to B'1'. In such a case, only user data is transmitted from the sender's IJBXSURS to the receiver's IJBXRUSR. The receiver can recognize such a condition by getting posted (IJBXRECB) and finding a zero data length value in the IJBXSLN field. Note, however, that the connection would still remain busy, also in this case, until the other side acknowledges via a RECEIVE or PURGE.

For performance reasons, the buffers should not cross page boundaries (if they are smaller than a page), and start on a page boundary (if they are larger than a page).

Data Transmission with Reply Request

Usually the sender posts a data transmission request to the receiver via SENDR.

The receiver accepts the data via RECEIVE, processes it, and sends a reply back to the sender via the REPLY request. The connection is now free to handle the next data transmission request.

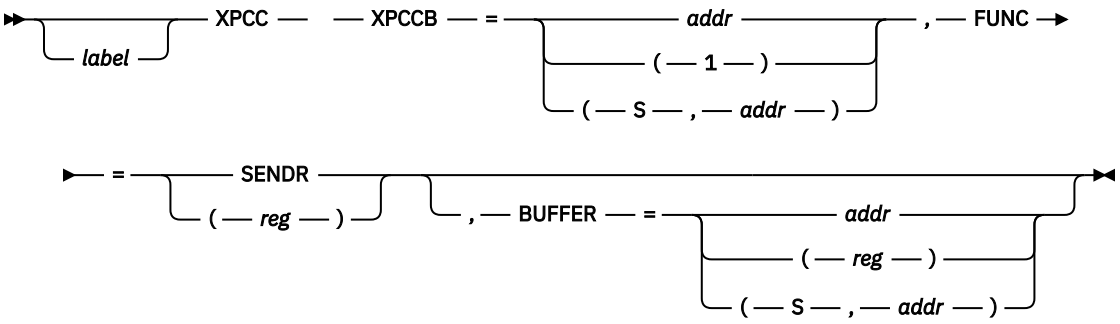


Table 15. REPLY request - Input fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXSNDR).
IJBXFDSC	If IJBXPOST is on, IJBXCECB will be posted when the other side receives the data. The buffers are free for usage from now.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXBUF	Buffer area for data transmission. The fields are used in the same way as described under SEND.
IJBXSUSR	Eight bytes of user data to be moved into receiver's IJBXRUSR field.
IJBXRADR	Address of area into which the system transfers the reply data from the receiver.
IJBXRLNG	Length of reply area.

Table 16. REPLY request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFDSC	Return codes set by system.
IJBXRETC	Return codes set by system.
IJBXSECB	Reset wait bit. Posted when the other side issues REPLY or PURGE.
IJBXCECB	Reset wait bit. Posted when the other side issues RECEIVE and if IJBXPOST was set.
IJBXRECB	Reset wait bit.
IJBXREAS	Set to IJBXRECX if IJBXPOST was set.

Table 17. REPLY request - Output fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXREAS	Reason codes.
IJBXRECB	The IJBXRECB is posted.
IJBXRUSR	8 bytes of data from the sender's IJBXSUSR field.
IJBXSLN	Length of data being sent. It is the sum of the length of all data areas.
IJBXSLNR	Length of reply data on sender's side.
IJBXFLG	Flag bytes. X'02' For a SENDR request, which means that a REPLY is requested.

Fields in the XPCCB used on the receiver's side:

The SENDR function is equivalent to the SEND function, except that in addition XPCC validates the area which will receive the reply (IJBXREPA) and moves the length of the REPLY area to IJBXSLNR at the receiver's side.

IJBXCECB at the sender's side is posted and the task is taken out of the wait state when the receiver issues RECEIVE (and IJBXREAS is set to IJBXRECX), if IJBXPOST was set on at SENDR time. (The sender may reuse the buffers from now on.)

IJBXSECB is posted and the task is taken out of the wait state when the receiver issues REPLY.

The SENDR function might be issued with the IJBXBLN field (length of data) initialized to zero and IJBXIND initialized to B'1xxx xxxx'. In such a case, only validation of the REPLY area is done and the user data is transmitted from the sender's IJBXSURS to the receiver's IJBXRUSR. The receiver can recognize such a condition by getting posted (IJBXRECB) and finding a zero data length value in the IJBXSLN field. He can immediately execute the REPLY (without being forced to execute a RECEIVE before REPLY).

Data Transmission into a Predefined Area

Via SENDI the sender moves data into the receive area defined at the receiver's side with the CONNECT request.

The length of the receive area can be retrieved from IJBXSILN of the sender's XPCCB after the CONNECT request was issued. When the SENDI is completed, the receiver is in SENDI state, which means that the current sender has to wait on a SENDI of his partner.

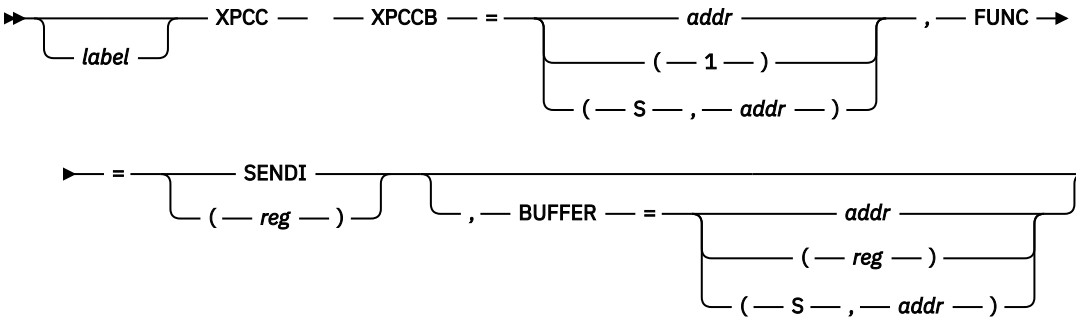


Table 18. SENDI request - Input fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXSNDI).
IJBXFDSC	Should be set to zero.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXBUF	Buffer area for data transmission. The fields are used in the same way as described under SEND.
IJBXSUSR	Eight bytes of user data to be moved into receiver's IJBXRUSR field.

Table 19. SENDI request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXSECB	Reset wait bit.
IJBXCECB	Reset wait bit.
IJBXRECB	Reset wait bit. Posted when other side issues SENDI.

Table 19. SENDI request - Output fields in the XPCCB used on the sender's side. (continued)	
Name	**Description**
IIJBXSLN	Length of data being sent.

Fields in the XPCCB used on the receiver's side:

Table 20. SENDI request - Output fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXREAS	IJBXSWSR is set if the receive area is too small.
IJBXRECB	The IJBXRECB is posted.
IJBXRUSR	8 bytes of data from the sender's IJBXSUSR field.
IJBXSLN	Length of data being sent. It is the sum of the length of all data areas.
IJBXFLG	Flag bytes. X'40' For a SENDI request, which means that a SENDI protocol is ongoing.

If the BUFFER operand is specified in the XPCC macro, it overwrites the IJBXBUF field in the XPCCB. The areas defined through IJBXBUF are validated and the length is checked against of the length of the partner's receive buffer specified with the CONNECT. Then the length is saved into IJBXSLN of the partner's XPCCB.

If the receive buffer is too small, XPCC switches to the SEND-RECEIVE protocol, that is, the receiver is posted with IJBXSWSR in IJBXREAS. He now has to issue a RECEIVE request to retrieve the data. The sender is notified by the protocol switch with return code 4 in register 15 and IJBXSSWI in IJBXRETC.

After execution of the SENDI request, the connection is free for another SENDI, but now from the current receiver.

Receiving Data

The target application recognizes a SEND request at the other side in the following way:

- IJBXRECB of the connection is posted.
- The IJBXCNTL field contains control information defining the SEND request.
- The IJBXFLG flag area contains a flag indicating whether this is a SEND or a SENDR request, or a SENDI request where the predefined receive area was too small.

With the RECEIVE request the application prompts the system for the actual data transfer.

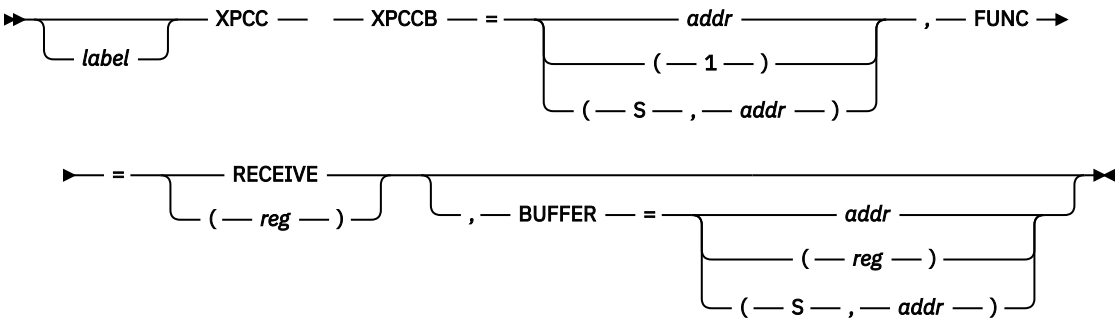


Table 21. RECEIVE request - Input fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXRCV).

Table 21. RECEIVE request - Input fields in the XPCCB used on the receiver's side. (continued)	
Name	**Description**
IJBXFDSC	Should be set to zero.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXBUF	Buffer area where the data is to be moved (only one area allowed: IJBXIND=B'1xxx xxxx').
IJBXSUSR	Eight bytes of user data to be moved into sender's IJBXRUSR field.

Table 22. RECEIVE request - Output fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXSECB	Reset wait bit.
IJBXCECB	Reset wait bit.
IJBXRECB	System resets the wait bit in case of a SEND. In case of a SENDR, the wait bit is reset at REPLY time.

Table 23. RECEIVE request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXRUSR	8 bytes of data from the receiver's IJBXSUSR field.
IJBXRSECB	Posted if this is a SEND-RECEIVE protocol.
IJBXCECB	Posted if this is a SENDR and IJBXPOST is on.
IJBXREAS	Set if this is a SENDR and IJBXPOST is on.
IJBXFLG	X'08' indicates RECEIVE was last XPCC function executed by other side.

A RECEIVE is requested when an application is posted by a SEND (SENDR) request.

If the XPCC FUNC=RECEIVE macro is used with the BUFFER operand, it overwrites the IJBXBUF field information stored in the XPCCB.

The program requests the data transfer via the RECEIVE request and XPCC moves the data into the input area.

If the RECEIVE is executed after a SEND request, IJBXRECB at the receiver's side is reset and the IJBXSECB at the sender's side is posted. The sender is also taken out of the wait state. The connection is now ready to handle the next SEND (or SENDR) request. Such a RECEIVE (or PURGE) is also needed in case of SEND with zero data in order to free the connection.

If the RECEIVE is executed after a SENDR request, the connection remains 'busy' until the receiver responds back to the sender via the REPLY request.

If the RECEIVE is due to SENDR (and IJBXPOST is on) or SEND, user data from the receiver's IJBXSUSR is transferred to IJBXRUSR of the sender's XPCCB; for SENDR, IJBXCECB is posted in addition.

The RECEIVE request performs an address validation of the receiver's data area. No padding is performed, if the data to be moved is shorter than the input area.

If the data to be moved is longer than the input area, the RECEIVE request is rejected with a return code. The program can then either obtain a longer input area and retry the RECEIVE request, or it can decide

that the incoming data block is too long to be handled and purge the connection from the sent data (refer to the PURGE function).

The REPLY Function

The REPLY function is used by the receiver when receiving data which was sent via a SENDR request.

It allows to send response data back to the sender into a predefined data area.

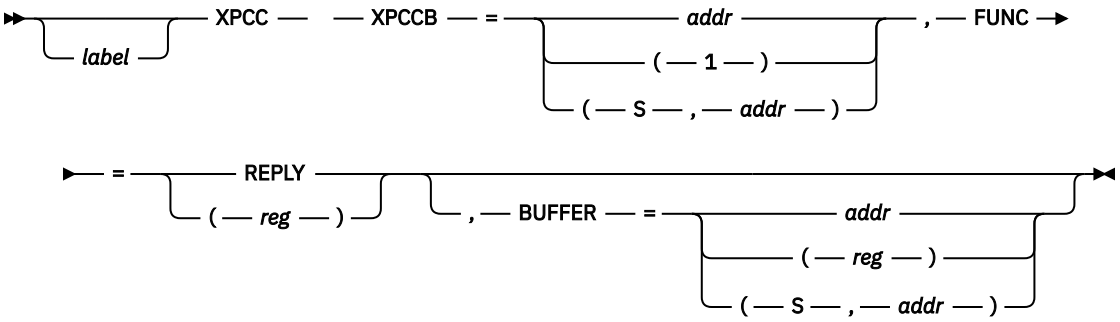


Table 24. REPLY request - Input fields in the XPCCB used on the replier's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXREP).
IJBXFDSC	Should be set to zero.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXBUF	Buffer area from where the data is transmitted. Only one buffer area allowed: IJBXIND=B'1'
IJBXSUSR	Eight bytes of user data to be moved into sender's IJBXRUSR field.

Table 25. REPLY request - Output fields in the XPCCB used on the replier's side.	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXRECB	Reset wait bit.

Table 26. REPLY request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXRUSR	8 bytes of data from the replier's IJBXSUSR field.
IJBXRSECB	Posted.
IJBXREAS	Reason code.
IJBXFLG	X'20' indicates last XPCC function executed by other side was REPLY.
IJBXSLN	Length of data being replied.

If the BUFFER operand is specified in the XPCC macro, it overwrites the IJBXBUF field in the XPCCB. The areas defined through IJBXBUF are validated and the length is checked against of the length of the partner's receive buffer specified with the CONNECT. Then the length is saved into IJBXSLN of the partner's XPCCB.

If the reply area length is 0, only the sender's IJBXSECB is posted and user data is transmitted from IJBXSUSR at the replier's side to IJBXRUSR at the sender's side. If the reply area on the sender's side is too long, the remaining bytes are not padded. If it is too short, the REPLY request is rejected.

IJBXRECB at the receiver's side is reset and IJBXSECB at the sender's side is posted. The sender is also taken out of the wait state.

Clearing a Pending SEND/SENDR Request on the Sender's Side

The sender of the data can revoke an outstanding SEND/SENDR request. For this purpose he can use the CLEAR function.

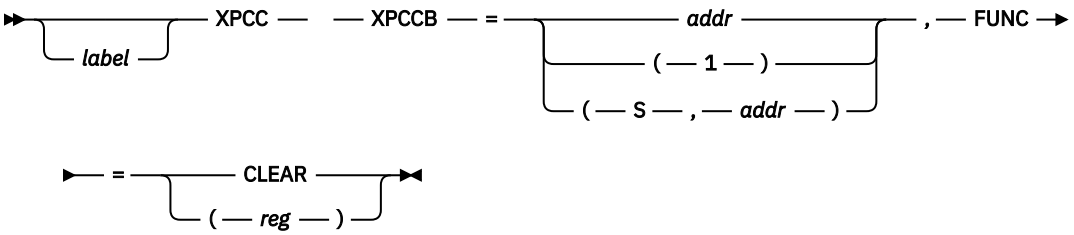


Table 27. CLEAR request - Input fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXCLR).
IJBXFDSC	Should be set to zero.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXSUSR	Eight bytes of user data to be moved into receiver's IJBXRUSR field.
IJBXCECB	Reset wait bit.
IJBXRECB	Reset wait bit.
IJBXSECB	Reset wait bit.

Table 28. CLEAR request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.

XPCCB fields used on the receiver's side:

Table 29. CLEAR request - Output fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXRUSR	8 bytes of data from the sender's IJBXSUSR field.
IJBXRSECB	The RECEIVE ECB is posted.
IJBXREAS	IJBXCLEA is posted to the reason code field.
IJBXFLG	X'04' indicates last XPCC function executed by the other side was CLEAR.

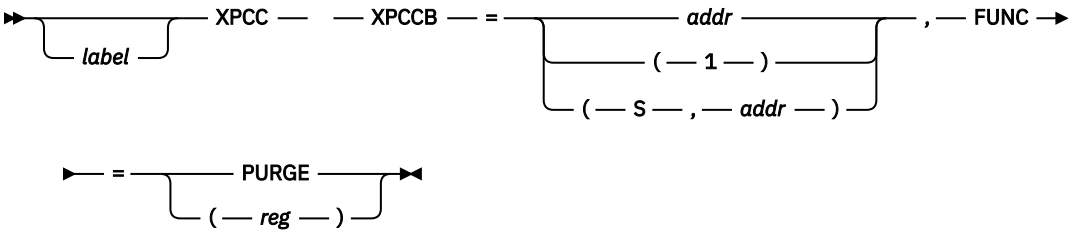
XPCC sets a 'SEND cleared' flag for the connection in the XPCCB, post IJBXRECB, and store reason code IJBXCLEA into IJBXREAS on the receiver's side in the following cases:

- If there is a SEND request pending for the connection, for which the other side has not yet issued a RECEIVE or PURGE.
- If there is a SENDR request pending for this connection, for which the other side has not yet issued the requested REPLY or PURGE.

To free the connection for the next SEND request, the receiver must issue an acknowledgment through RECEIVE or PURGE. RECEIVE/PURGE will complete with a return code indicating a cleared connection. The connection is then ready for the next SEND (SENDR) request.

Clearing a Pending SEND/SENDR Request on the Receiver's Side

The receiver might receive messages which he is unable to handle. For example, because the message length exceeds the available buffer storage. He can reject those messages via the PURGE request.



XPCCB fields used at the receiver's side:

Table 30. PURGE request - Input fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXFCT	Function byte (IJBXPRG).
IJBXFDSC	Should be set to zero.
IJBXPID	Path ID, provided by the system after CONNECT.
IJBXSUSR	Eight bytes of user data to be moved into sender's IJBXRUSR field.

Table 31. PURGE request - Output fields in the XPCCB used on the receiver's side.	
Name	**Description**
IJBXFDSC	Set to X'00'.
IJBXRETC	Return code.
IJBXSECB	Reset wait bit.
IJBXCECB	Reset wait bit.
IJBXRECB	Reset wait bit.

Table 32. PURGE request - Output fields in the XPCCB used on the sender's side.	
Name	**Description**
IJBXRUSR	8 bytes of data from the receiver's IJBXSUSR field.
IJBXRSECB	Posted.
IJBXCECB	Posted after SENDR and if IJBXPOST is on.
IJBXREAS	Appropriate reason code is set.
IJBXFLG	X'10' Indicates that last XPCC function executed by the other side was PURGE.

XPCC clears the connection from the pending SEND/SENDR request. The receiver's IJBXRECB is reset. At the sender's side the IJBXSECB is posted with reason code IJBXCPRG indicating the PURGE request, and the sender is taken out of the wait state. Additionally, IJBXCECB is posted when a SENDR was pending and IJBXPOST was on.

With PURGE, the receiver can also acknowledge a CLEAR request from the sender. In this case no reason code is posted back.

Disconnecting from a Communication Path

If a data communication path is not needed any more, an application can break it via the DISCONNECT function. When the other side disconnects from this path, a DISCONNECT is also required from this application.

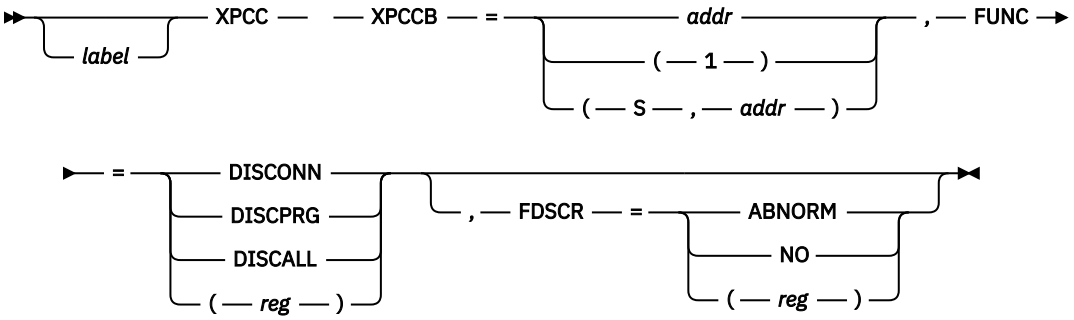


Table 33. DISCONNECT request - Input fields in the XPCCB	
Name	**Description**
IJBXFCT	Function byte (IJBXDSC|IJBXDSCP|IJBXDSCA).
IJBXFDSC	If FDSCR=ABNORM is specified in the XPCC request, the other side gets reason code IJBXABDC indicating an abnormal-end condition.
IJBXIDK	Identify token provided by the system after IDENT of requesting application. This field is only used with DISCALL.
IJBXPID	Path ID, provided by the system after CONNECT. This field is not used for a DISCALL request.
IJBXSUSR	Eight bytes of user data to be moved into field IJBXRUSR of the other side. Not used with DISCALL.

Table 34. DISCONNECT request - Output fields in the XPCCB	
Name	**Description**
IJBXRETC	Return code.

Table 35. DISCONNECT request - Output fields in the XPCCB used on the partner's side.	
Name	**Description**
IJBXRUSR	8 bytes of data from IJBXSUSR of the other side. Only used for the first DISCONNECT,because with the second DISCONNECT the other side is already gone.
IJBXCECB	Posted.
IJBXSECB	Posted.
IJBXRECB	Posted.

Table 35. DISCONNECT request - Output fields in the XPCCB used on the partner's side. (continued)	
Name	**Description**
IJBXREAS	Appropriate reason code is set.

DISCONN
Checks whether the link is still busy. If so, rejects the request with a return code. If not, disconnects the link on the requester's side.

DISCPRG
Disconnects the link unconditionally, regardless of whether the link is still busy or not. If the other side has an outstanding SEND or SENDR request on this link, the request is purged and IJBXSECB is posted, together with a reason code. If an own SEND is pending, this send request is cleared.

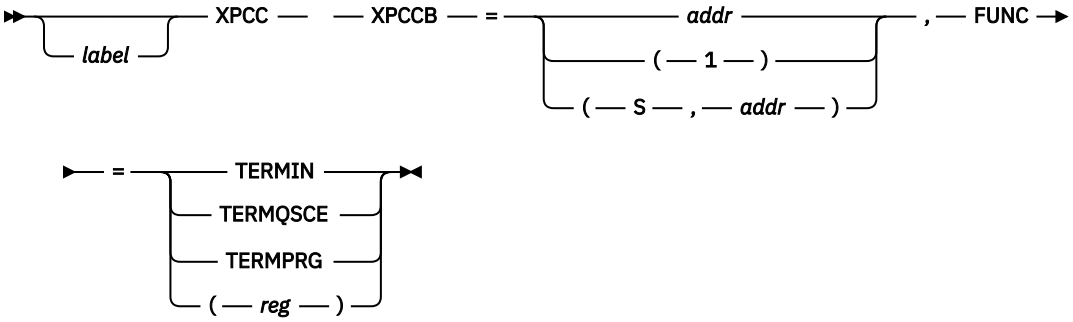
DISCALL
Unconditionally disconnects all connections set up by the corresponding application. It can only be issued by the task that issued the corresponding IDENT request. The DISCALL function implies a CLEAR or PURGE, if necessary.

If the other side is in the wait state at the time the request is issued, it will be posted (IJBXCECB, IJBXSECB, and IJBXRECB), and a reason code will be set.

FDSCR=ABNORM
This option can be specified with DISCONN and DISCPRG. In this case XPCC posts an abnormal-end condition to the partner (IJBXABDC in IJBXREAS).

Terminating XPCC Usage

If an application does not need the XPCC service any longer, it can 'log off' from XPCC by issuing a TERMINATE request.



XPCCB fields used by the system:

Table 36. TERMINATE request - Input fields in the XPCCB used by the system	
Name	**Description**
IJBXFCT	Function byte (IJBXTRM|IJBXTRMP|IJBXTRMQ).
IJBXFDSC	Should be set to zero.
IJBXPIDK	Identify token provided by the system after IDENT of requesting application.

Table 37. TERMINATE request - Output fields in the XPCCB used by the system	
Name	**Description**
IJBXRETC	Return code.

XPCCB fields used in XPCCBs :

Table 38. TERMQSCE request - Output fields in the XPCCB, which have CONNECTs for terminating applications pending (TERMQSCE only).	
Name	**Description**
IJBXCECB	Posted.
IJBXSECB	Posted.
IJBXRECB	Posted.
IJBXREAS	Return codes.

TERMIN
Checks if any connections still exist for this application. If yes, the request is rejected with a return code. If not, XPCC purges all information about this application.

TERMPRG
Unconditionally executes the terminate request. All available links are unconditionally disconnected and all pending data requests terminated (via DISCALL).

TERMQSCE
The application indicates that it is shortly going to perform a shut-down operation. Existing connections can still be used for data transmission. However, XPCC will no longer grant any new CONNECT request to/from this application.

All still 'open-ended' connections from this application are disconnected.

All open CONNECT requests for this application are set ready, all three ECBs are posted, and IJBXREAS is set to IJBXDISC.

Abnormal End Processing

For clean-up purposes, the system associates IDENT and CONNECT requests with certain 'work units', which it knows:

1. An IDENT request can be issued either by a VSE maintask or under control of a subtask. In the first case, the IDENT is regarded as being owned by the VSE partition. If this partition terminates, the system issues a TERMPRG for this IDENT request (if not already terminated).

   In the second case, the IDENT is regarded as being owned by the VSE subtask. If this subtask terminates, the system will issue a TERMPRG for this IDENT request.
2. Each CONNECT request is associated with a VSE task, under whose control the CONNECT was requested.

   If the task terminates, the system will disconnect all connections that were set up by this task.

If subsystems are using smaller work units for their applications, they have to do the DISCONNECT and TERMINATE requests for the ABENDed application ('private' subtasking).

Compressing and Expanding Data

This section contains information on the z/VSE support for compression services. The compression services allow you to compress data and expand data that was previously compressed. The interface to the compression services is the CSRCMPSC macro. The CSRCMPSC macro uses the CMPSC hardware instruction, if available; otherwise, the CSRCMPSC macro simulates the CMPSC instruction.

You can save data in a compressed format, for example to conserve DASD resources. The CSRCMPSC macro provides a pair of services that compress and expand data. These services are available when bit CVTCMPSC in the communication vector table (CVT) is on.

Compression takes an input string of data and, using a data area called a dictionary, produces an output string of compression symbols. Each symbol represents a string of one or more characters from the input.

Expansion takes an input string of compression symbols and, using a dictionary, produces an output string of the characters represented by those compression symbols.

Parameters for the CSRCMPSC macro are in an area mapped by DSECT CMPSC of the CSRYCMPS macro and specified by the CBLOCK parameter of the CSRCMPSC macro. This area contains such information as:

- The address, ALET, and length of a source area. The source area contains the data to be compressed for a compression operation, or to be expanded for an expansion operation.
- The address, ALET, and length of a target area. After the macro runs, the target area contains the compressed data for a compression operation, or the expanded data for an expansion operation.
- An indication of whether to perform compression or expansion.
- The address and format of a dictionary to be used to perform the compression or expansion. The dictionary must be in the same address space as the source area.

Compressing and expanding data is described in the following topics:

- "Compression and Expansion Dictionaries" on page 166
- "Compression Processing" on page 167
- "Expansion Processing" on page 167
- "Dictionary Entries" on page 168
- "Building the CSRYCMPS Area" on page 176
- "Determining if the CSRCMPSC Macro Can Be Issued on a System" on page 178

If you are a z/OS (MVS) user, to help you use the compression services, the SYS1.SAMPLIB system library contains the following REXX execs:

- CSRBDICT for building example dictionaries
- CSRCMPEX for measuring the degree of compression that the dictionaries provide

The prologs of the execs tell how to use them. For additional information about compression and using the execs, see *Enterprise Systems Architecture/390 Data Compression* and the *Principles of Operation* publication for your processor.

Compression and Expansion Dictionaries

To accomplish compression and expansion, the macro uses two dictionaries: the compression dictionary and the expansion dictionary. These dictionaries are related logically and physically. When you expand data that has been compressed, you want the result to match the original data. Thus the dictionaries are complementary. When compression is done, the expansion dictionary must immediately follow the compression dictionary, because the compression algorithm examines entries in the expansion dictionary.

Each dictionary consists of 512, 1024, 2048, 4096, or 8192 8-byte entries and begins on a page boundary. When the system determines or uses a compression symbol, the symbol is 9, 10, 11, 12, or 13 bits long, with the length corresponding to the number of entries in the dictionary. Specify the size of the dictionary in the CMPSC_SYMSIZE field of the CSRYCMPS mapping macro:

SYMSIZE
Meaning

1
Symbol size 9 bits, dictionary has 512 entries

2
Symbol size 10 bits, dictionary has 1024 entries

3
Symbol size 11 bits, dictionary has 2048 entries

4
Symbol size 12 bits, dictionary has 4096 entries

5
Symbol size 13 bits, dictionary has 8192 entries

The value of CMPSC_SYMSIZE represents the size of the compression and expansion dictionaries. For example, if CMPSC_SYMSIZE is 512, then the size of the compression dictionary is 512 and the size of the expansion dictionary is 512.

Compression Processing

The compression dictionary consists of a specified number of 8-byte entries. The first 256 dictionary entries correspond to the 256 possible values of a byte and are referred to as *alphabet entries*. The remaining entries are arranged in a downward tree, with the alphabet entries being the topmost entries in the tree. That is, an alphabet entry may be a *parent entry* and contain the index of the first of one or more contiguous *child entries*. A child entry may, in turn, be a parent and point to its own children. Each entry may be identified by its *index*, meaning the positional number of the entry in the dictionary; the first entry has an index of 0.

An alphabet entry represents one character. A nonalphabet entry represents all of the characters represented by its ancestors and also one or more additional characters called *extension characters*. For compression, the system uses the first character of an input string as an index to locate the corresponding alphabet entry. Then the system compares the next character or characters of the string against the extension character or characters represented by each child of the alphabet entry until a match is found. The system repeats this process using the children of the last matched entry, until the last possible match is found, which might be a match on only the alphabet entry. The system uses the index of the last matched entry as the compression symbol.

The first extension character represented by a child entry exists as either a child character in the parent or as a sibling character. A parent can contain up to four or five child characters. If the parent has more children than the number of child characters that can be in the parent, a dictionary entry named a *sibling descriptor* follows the entry for the last child character in the parent. The sibling descriptor can contain up to six additional child characters, and a dictionary entry named a sibling descriptor extension can contain eight more child characters for a total of fourteen. These characters are called *sibling characters*. The corresponding additional child entries follow the sibling descriptor. If necessary, another sibling descriptor follows the additional child entries, and so forth. The dictionary entries that are not sibling descriptors or sibling descriptor extensions are called character entries.

If a nonalphabet character entry represents more than one extension character, the extension characters after the first are in the entry; they are called additional extension characters. The first extension character exists as a child character in the parent or as a sibling character in a sibling descriptor or sibling descriptor extension. The nonalphabet character entries represent either:

- If the entry has no children or one child, from one to five extension characters.
- If the entry has more than one child, one or two extension characters. If the entry represents one extension character, it can contain five child characters. If it represents two extension characters, it can contain four child characters.

Expansion Processing

The dictionary used for expansion also consists of a specified number of 8-byte entries. The two types of entries used for expansion are:

- Unpreceded entries
- Preceded entries

The compression symbol, which is an index into the dictionary, locates that index's dictionary entry. The symbol represents a character string of up to 260 characters. If the entry is an unpreceded entry, the expansion process places at offset 0 from the current processing point the characters designated by that entry. Note that the first 256 correspond to the 256 possible values of a byte and are assumed to designate only the single character with that byte value.

If the entry is a preceded entry, the expansion process places the designated characters at the specified offset from the current processing point. It then uses the information in that entry to locate the preceding entry, which may be either an unpreceded or a preceded entry, and continues as described previously.

The sibling descriptor extension entries described earlier are also physically located within the expansion dictionary.

Dictionary Entries

The following notation is used in the diagrams of dictionary entries:

{cc}
Character may be present

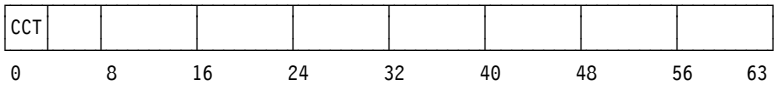
...
The preceding field may be repeated

Compression Dictionary Entries

Compression entries are mapped by DSECTs in macro CSRYCMPD.

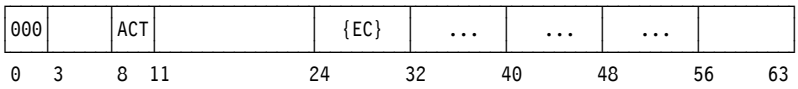
The first four entries that follow give the possible values for bits 0-2, which are designated CCT.

Character Entry Generic Form (DSECT CMPSCDICT_CE)



CCT
A 3-bit field (CMPSCDICT_CE_CHILDCT) specifying the number of children. The total number of children plus additional extension characters is limited to 5. If this field plus the number of additional characters is 6, it indicates that, in addition to the maximum number of children for this entry, there is a sibling descriptor entry that describes additional children. The sibling descriptor entry is located at dictionary entry CMPSCDICT_CE_FIRSTCHILDINDEX plus the value of CMPSCDICT_CE_CHILDCT. The value of CMPSCDICT_CE_CHILDCT plus the number of additional extension characters must not exceed 6.

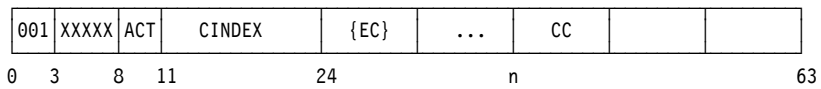
Character Entry CCT=0 (DSECT CMPSCDICT_CE)



ACT
A 3-bit field (CMPSCDICT_CE_AECCT) indicating the number of additional extension characters in the entry. Its value must not exceed 4. This field must be 0 in an alphabet entry.

EC
An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

Character Entry CCT=1 (DSECT CMPSCDICT_CE)



XXXXX
A 5-bit field (CMPSCDICT_CE_EXCHILD) with the first bit indicating whether it is necessary to examine the character entry for the child character (looking either for additional extension characters or more children). The other bits are ignored when CCT=1.

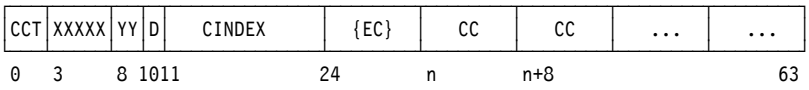
ACT
A 3-bit field (CMPSCDICT_CE_AECCT) indicating the number of additional extension characters. Its value must not exceed 4. This field must be 0 in an alphabet entry.

CINDEX
A 13-bit field (CMPSCDICT_CE_FIRSTCHILDINDEX) indicating the index of the first child. The index for child *n* is then CMPSCDICT_CE_FIRSTCHILDINDEX + n-1.

EC
An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

CC
Child character, at bit n = 24 + (ACT * 8). The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters.

Character Entry CCT>1 (DSECT CMPSCDICT_CE)



CCT
A 3-bit field (CMPSCDICT_CE_CHILDCT) specifying the number of children. For this case, because CCT>1, the range for CCT is 2 to 6 if D=0 or 2 to 5 if D=1. If this field plus the value of D is 6, it indicates that, in addition to the maximum number of children for this entry (4 if D=1, 5 if D=0), there is a sibling descriptor entry that describes additional children. The sibling descriptor entry is located at dictionary entry CMPSCDICT_CE_FIRSTCHILDINDEX plus the value of CMPSCDICT_CE_CHILDCT.

XXXXX
A 5-bit field (CMPSCDICT_CE_EXCHILD) with a bit for each child in the entry. The field indicates whether it is necessary to examine the character entry for the child character (looking either for additional extension characters or more children). The bit is ignored if the child does not exist.

YY
A 2-bit field (CMPSCDICT_CE_EXSIB) providing examine-child bits for the 13th and 14th siblings designated by the first sibling descriptor for children of this entry. The bit is ignored if the child does not exist. Note that this is a subfield of CMPSCDICT_CE_AECCT. Do not set both this field and field CMPSCDICT_CE_AECCT in a character entry.

D
A 1-bit field (CMPSCDICT_CE_ADDEXTCHAR) indicating whether there is an additional extension character. Note that this is a subfield of CMPSCDICT_CE_AECCT. Do not set both this field and field CMPSCDICT_CE_AECCT in a character entry. This bit must be 0 in an alphabet entry.

CINDEX
A 13-bit field (CMPSCDICT_CE_FIRSTCHILDINDEX) indicating the index of the first child. The index for child *n* is CMPSCDICT_CE_FIRSTCHILDINDEX + n-1.

EC
An additional extension character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension character followed by the child characters. There is no additional extension character if D=0.

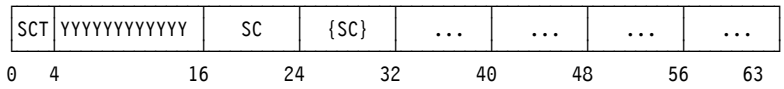
CC
Child character. The 5-character field CMPSCDICT_CE_CHILDCHAR is provided to hold the additional extension characters followed by the child characters. The first child character is at bit n = 24 + (D * 8).

Alphabet Entries (DSECT CMPSCDICT_CE)

The alphabet entries have the same mappings as character entries but without the additional extension characters. The character entries are “Compression Dictionary Entries” on page 168, “Character Entry

CCT=0 (DSECT CMPSCDICT_CE)" on page 168, "Character Entry CCT=1 (DSECT CMPSCDICT_CE)" on page 168, and "Character Entry CCT>1 (DSECT CMPSCDICT_CE)" on page 169.

Format 1 Sibling Descriptor (DSECT CMPSCDICT_SD)



SCT
A 4-bit field (CMPSCDICT_SD_SIBCT) specifying the number of sibling characters. The number of sibling characters is limited to 14. If this field is 15, it indicates that there are 14 sibling characters associated with this entry and that there is another sibling descriptor entry, which describes additional children. That sibling descriptor entry is located at dictionary entry this-sibling-descriptor-index + 15. If there are 1 to 6 sibling characters, they are contained in this entry, and the dictionary entries for those characters are located at this-sibling-descriptor-index + n, where n is 1 to 6. If there are 7 to 14 sibling characters, the first 6 are as described above, and characters 7 through 14 are located in the expansion dictionary entry. (See "Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)" on page 171.) The index of the character entry is this-sibling-descriptor-index. The number of sibling characters should not be 0.

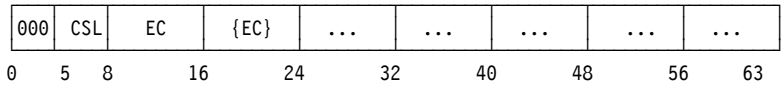
YYYYYYYYYYYY
A 12-bit field (CMPSCDICT_SD_EXSIB), one for each sibling character, indicating whether to examine the character entries for sibling characters 1 through 12. Recall that the examine-sibling indicator for sibling characters 13 and 14 for the first sibling descriptor is in the character entry field CMPSCDICT_CE_EXSIB. If this is not the first sibling descriptor for the child entry, then the character entries for sibling characters 13 and 14 are examined irregardless. The bit is ignored if the sibling does not exist.

SC
Sibling character. Sibling characters 8 through 14 are in the expansion dictionary. (See "Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)" on page 171.) The 6-character field (CMPSCDICT_SD_CHILDCHAR) is provided to contain the sibling characters. The index of the character entry for sibling character *n* is this-sibling-descriptor-index + n-1.

Expansion Dictionary Entries

Expansion entries are mapped by DSECTs in macro CSRYCMPD.

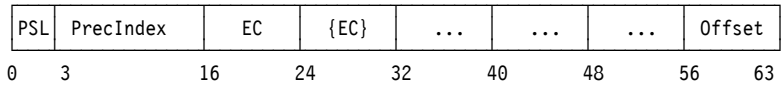
Unpreceded Entry (DSECT CMPSCDICT_UE)



CSL
A 3-bit field (CMPSCDICT_UE_COMPSYMLEN) indicating the number of characters contained in CMPSCDICT_UE_CHARS. These characters will be placed at offset 0 in the expanded output. This field should not have a value of 0.

EC
Expansion character. The 7-character field (CMPSCDICT_UE_CHARS) is provided to contain the expansion characters.

Preceded Entry (DSECT CMPSCDICT_PE)



PSL
A 3-bit field (CMPSCDICT_PE_PARTSYMLEN) indicating the number of characters contained in CMPSCDICT_PE_CHARS. These characters will be placed at the offset indicated by CMPSCDICT_PE_OFFSET in the expanded output. This field must not be 0, because 0 indicates an unpreceded entry.

PrecIndex
A 13-bit field (CMPSCDICT_UE_PRECENTINDEX) indicating the index of the dictionary entry with which processing is to continue.

EC
Expansion character. The 5-character field (CMPSCDICT_PE_CHARS) is provided to contain the expansion characters.

Offset
A 1-byte field (CMPSCDICT_PE_OFFSET) indicating the offset in the expanded output for characters in CMPSCDICT_PE_CHARS.

Sibling Descriptor Extension Entry (DSECT CMPSCDICT_SDE)

SC
Sibling character. The 8-character field (CMPSCDICT_SDE_CHARS) is provided to contain the sibling characters. The nth sibling character in this entry is actually overall sibling character number 6 + n, because the first 6 characters were contained in the corresponding sibling descriptor entry. The index of the character entry for the nth character is this-sibling-descriptor-index + 6 + n-1.

Dictionary Restrictions

Set up the compression dictionary so that:

- The algorithm does not create a compression symbol that represents a string of more than 260 characters.
- No character entry has more than 260 total children, including all sibling descriptors for that character entry.
- No character entry has a child count greater than 6.
- No character entry has more than 4 additional extension characters when there are 0 or 1 child characters.
- No sibling descriptor indicates 0 sibling characters.

Set up the expansion dictionary so that:

- Expansion of a compression symbol does not use more than 127 dictionary entries.

Other Considerations

If the first child character matches, but its additional extension characters do not match and the next child character is the same as the first, the system continues compression match processing to try to find a compression symbol that contains that child character. If, however, the next child character is not the same, compression processing uses the current compression symbol as the result. You can set up the child characters for an entry to take advantage of this processing.

If a parent entry does not have the examine child bit (CMPSCDICT_CE_EXCHILD) on for a particular child character, then the child character entry should not have any additional extension characters or children. The system will not check the entry itself for additional extension characters or children.

If a parent or sibling descriptor entry does not have the examine sibling bit (CMPSCDICT_CE_EXSIB) on for a particular sibling character, then the character entry for that sibling character should not to have

any additional extension characters or children. The system will not check the entry itself for additional extension characters or children.

Compression Dictionary Examples

In the following examples, most fields contain their hexadecimal values. However, for clarity, the examine-child bit fields are displayed with their bit values.

Example 1

Suppose the dictionary looks like the following:

Hexadecimal Entry
Description

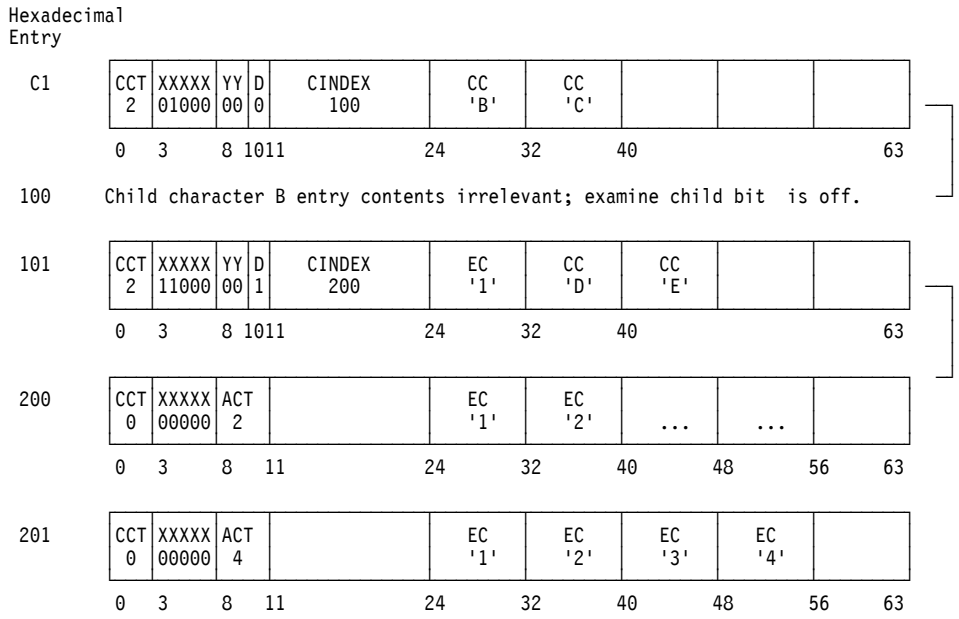
C1
Alphabet entry for character A; 2 child characters B and C. The first child index is X'100'.

100
Entry for character B; no additional extension characters; no children.

101
Entry for character C; additional extension character 1; 2 child characters D and E. The first child index is X'200'.

200
Entry for character D; 2 additional extension characters 1 and 2; no children.

201
Entry for character E; 4 additional extension characters 1, 2, 3, and 4; no children.



If the input string is AD, the output string will consist of 2 compression symbols: one for A and one for D. When examining the dictionary entry for character A, the system determines that none of A's children match the next input character, D, and returns the compression symbol for A. When examining the dictionary entry for character D, the system determines that it has no children, and so returns the compression symbol for D.

If the input string is AB, the output string will consist of 1 compression symbol for both input characters. When examining the dictionary input for character A, the system determines that A's first child character matches the next input character, B, and looks at entry X'100'. Because that entry has no additional extension characters, a match is determined. Because there are no further input characters, the scan concludes.

If the input string is AC, the output string will consist of 2 compression symbols: one for A and one for C. When examining the dictionary input for character A, the system determines that A's second child character matches the next input character, C, and looks at entry X'101'. Because that entry has an additional extension character, but the input string does not contain this character, no match is made, and the output is the compression symbol for A. Processing character C results in the compression symbol for C.

If the input string is AC1, the output string will consist of 1 compression symbol. When examining the dictionary input for character A, the system determines that A's second child character matches the next input character, C, and looks at entry X'101'. Because that entry has an additional extension character, and the input string does contain this character, 1, a match is made, and the output is the compression symbol for AC1.

Similarly, the set of input strings longer than one character compressed by this dictionary are:

Hexadecimal Symbol
String

100
AB

101
AC1

200
AC1D12

201
AC1E1234

The compression symbol is the index of the dictionary entry. Based on this, you can see that the expansion dictionary must result in the reverse processing; for example, if a compression symbol of X'201' is found, the output must be the string AC1E1234. See “Expansion Dictionary Example” on page 175 for expansion dictionary processing.

Example 2 for More than 5 Children

Suppose the dictionary looks like the following:

Hexadecimal Entry
Description

C2
Alphabet entry for character B; child count of 6 (indicating 5 children plus a sibling descriptor); first child index is X'400', children are 1, 2, 3, 4, and 5.

400
Entry for character 1; no additional extension characters; no children.

401-404
Entries for characters 2 through 5; no additional extension characters; no children.

405
Sibling descriptor; child count of 15, which indicates 14 children plus another sibling descriptor; sibling characters A, B, C, D, E, and F.

405
Sibling descriptor extension. In the expansion dictionary entry X'405', the sibling characters are G, H, I, J, K, L, M, and N.

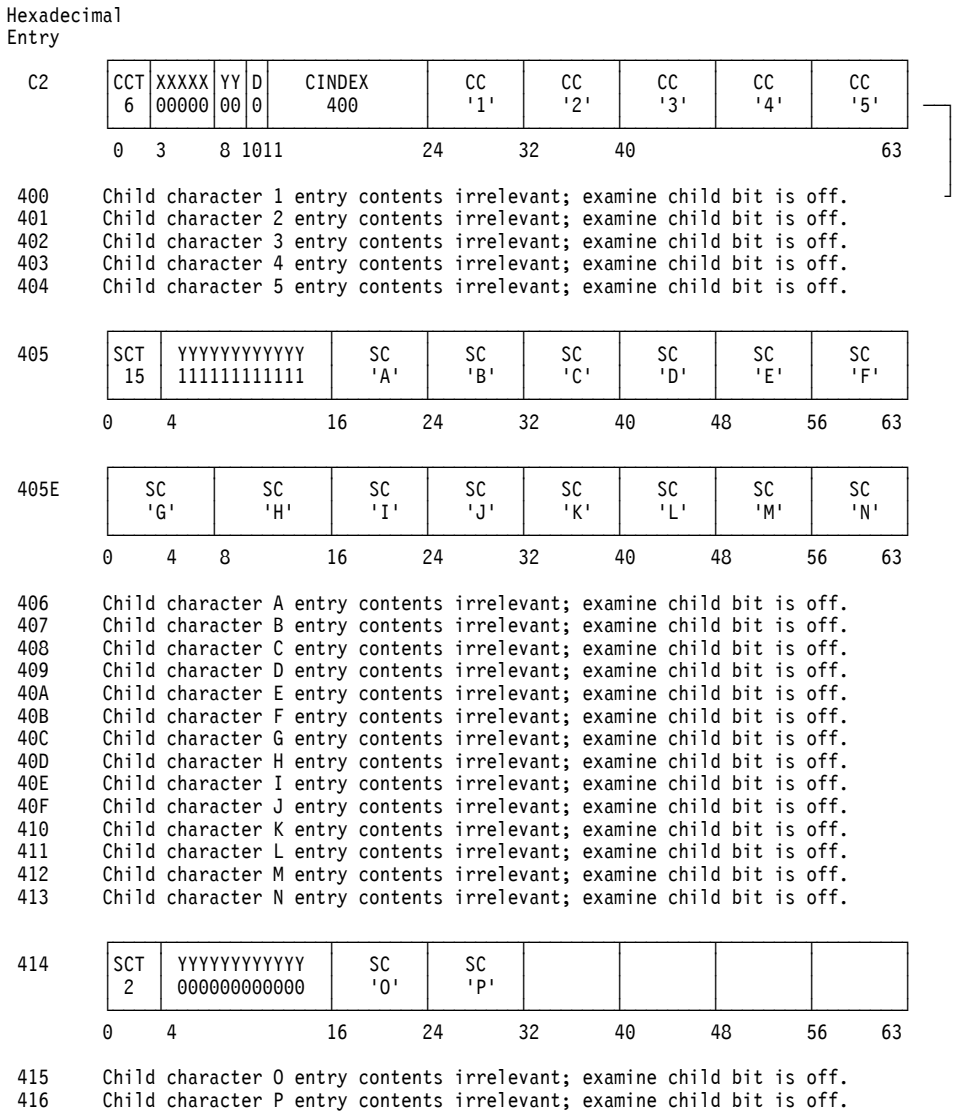
406
Entry for character A; no additional extension characters; no children.

407-413
Entries for characters B through N; no additional extension characters; no children.

414
Next sibling descriptor; child count of 2; child characters O and P.

415
Entry for character O; no additional extension characters; no children.

416
Entry for character P; no additional extension characters; no children.



The set of input strings longer than one character compressed by this dictionary are:

Hexadecimal Symbol
String

400-404
B1, B2, B3, B4, B5

406-40B
BA, BB, BC, BD, BE, BF

40C-413
BG, BH, BI, BJ, BK, BL, BM, BN

415-416
BO, BP

There are no compression symbols for 405 and 414. These are the sibling descriptor entries. Because their sibling descriptor extensions are located at those indices in the expansion dictionary (not the preceded or unpreceded entries required for expansion), it is important that no compression symbol have that value.

Example 3 for Children with the Same Value

Suppose the dictionary looks like the following:

Hexadecimal Entry
Description

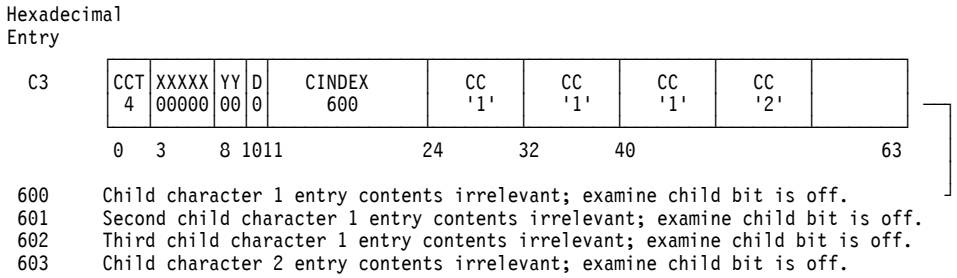
C3
Alphabet entry for character C; child count of 4. The first child index is X'600' and the child characters are 1, 1, 1, and 2.

600
Entry for character 1; 4 additional extension characters A, B, C, and D; no children.

601
Entry for character 1; 3 additional extension characters A, B, and C; no children.

602
Entry for character 1; 2 additional extension characters A and B; no children.

603
Entry for character 2; no additional extension characters; no children.



The set of input strings longer than one character compressed by this dictionary are:

Hexadecimal Symbol
String

600
C1ABCD

601
C1ABC

602
C1AB

603
C2

By taking advantage of the special processing when the second and subsequent child characters match the first, you can reduce the number of dictionary entries searched to determine the compression symbols. For example, to find that X'601' is the compression symbol for the characters C1ABC, the processing examines entry X'C3', then entry X'600' then entry X'601' Entry X'600' does not match because the input string does not have all 4 extension characters. There are alternate ways of setting up the dictionary to compress the same set of input strings handled by this dictionary.

Expansion Dictionary Example

Example

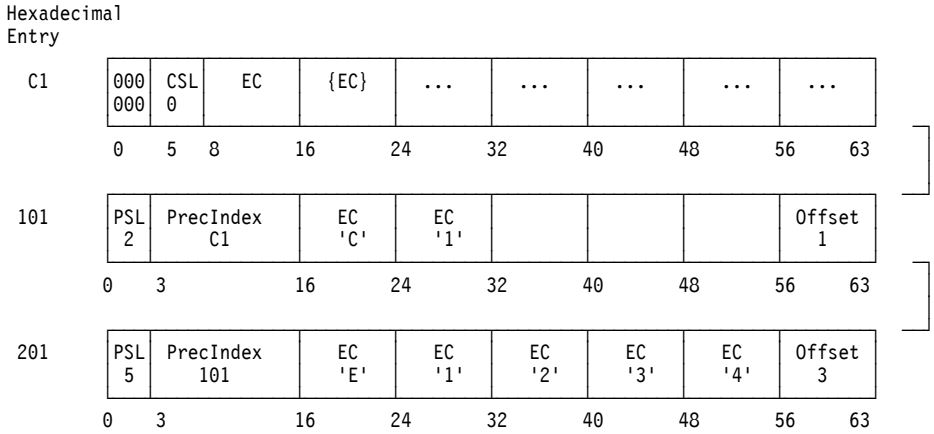
Suppose the expansion dictionary looks like the following:

Hexadecimal Entry
Description

C1
Alphabet entry for character A. This by definition is an unpreceded entry.

101
A preceded entry, with characters C and 1; with preceding entry index of X'C1'; offset of 1.

201
A preceded entry, with characters E, 1, 2, 3, and 4; with preceding entry index of X'101'; offset of 3.



When processing an input compression symbol of X'201':

- Characters E1234 are placed at offset 3, and processing continues with entry X'101'.
- Characters C1 are placed at offset 1, and processing continues with entry X'C1'.
- Character A is placed at offset 0.

The expansion results in the 8 characters A, C, 1, E, 1, 2, 3, and 4 placed in the output string.

Building the CSRYCMPS Area

The CSRYCMPS area is mapped by the CSRYCMPS mapping macro and is specified in the CBLOCK parameter of the CSRCMPSC macro. The area consists of 7 words that should begin on a word boundary. Unused bits in the first word must be set to 0.

- Set 4-bit field CMPSC_SYMSIZE in byte CMPSC_FLAGS_BYTE2 to a number from 1 to 5 to indicate both the number of entries in the dictionary and the size of a compressed symbol.
- If expanding, turn on bit CMPSC_EXPAND in byte CMPSC_FLAGS_BYTE2. Otherwise, make sure that the bit is off.
- Set field CMPSC_DICTADDR to the address of the necessary dictionary. If compressing, this should be the compression dictionary, which must be immediately followed by the expansion dictionary. If expanding, this should be the expansion dictionary. In either case, the dictionary must begin on a page boundary, as the low order 12 bits of the address are assumed to be 0 when determining the address of the dictionary.

  If running in AR mode, set field CMPSC_SOURCEALET to the ALET of the necessary dictionary. Note that the input area is also accessed using this ALET. If not in AR mode, make sure that the field contains 0.
- In most cases, make sure that 3-bit field CMPSC_BITNUM in byte CMPSC_DICTADDR_BYTE3 is zero. This field has the following meaning:
  - If compressing, place the first compression symbol at this bit in the leftmost byte of the target operand. Normally this field should be set to 0 for the start of compression.
  - If expanding, expand beginning with the compression symbol that begins with this bit in the leftmost byte of the source operand. Normally this field should be set to the value used for the start of compression.
- Set word CMPSC_TARGETADDR to the address of the output area. For compression, the output area contains the compressed data; for expansion, it contains the expanded data.

If running in AR mode, set field CMPSC_TARGETALET to the ALET of the output area. If not in AR mode, make sure that the field contains 0.

- Set word CMPSC_TARGETLEN to the length of the output area.
- Set word CMPSC_SOURCEADDR to the address of the input area. For compression, the input area contains the data to be compressed; for expansion, it contains the compressed data.

  If running in AR mode, set field CMPSC_SOURCEALET to the ALET of the input area. Note that the dictionary will also be accessed using this ALET. If not in AR mode, make sure that the field contains 0.
- Set word CMPSC_SOURCELEN to the length of the input area. For expansion, the length should be the difference between CMPSC_TARGETLEN at the completion of compression and CMPSC_TARGETLEN at the start of compression, increased by 1 if field CMPSC_BITNUM was nonzero upon completion of compression.
- Set word CMPSC_WORKAREAADDR to the address of a 192-byte work area for use by the CSRCMPSC macro. The work area should begin on a doubleword boundary. This area does not need to be provided and the field does not have to be set if your code has verified that the hardware CMPSC instruction is present. The program can do the verification by checking that bit CVTCMPSH in mapping macro CVT is on.

When the CSRCMPSC service returns, it has updated the input CSRYCMPS area as follows:

- CMPSC_FLAGS is unchanged.
- CMPSC_DICTADDR is unchanged, but bits CMPSC_BITNUM in field CMPSC_DICTADDR_BYTE3 are set according to the last-processed compression symbol.
- CMPSC_TARGETADDR is increased by the number of output bytes processed.
- CMPSC_TARGETLEN is decreased by the number of output bytes processed.
- CMPSC_SOURCEADDR is increased by the number of input bytes processed.
- CMPSC_SOURCELEN is decreased by the number of input bytes processed.
- CMPSC_WORKAREA is unchanged.

The target/source address and length fields are updated analogously to the corresponding operands of the MVCL instruction, so that you can tell upon completion of the operation how much data was processed and where you might want to resume if you wanted to continue the operation.

Suppose that you had compressed a large area but wanted to expand it back into a small area of 80-byte records. You might do the expansion as follows:

```
         LA   2,MYCBLOCK
         USING CMPSC,2
         XC   CMPSC(CMPSC_LEN),CMPSC
         OI   CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_1
         OI   CMPSC_FLAGS_BYTE2,CMPSC_EXPAND
         L     3,EDICTADDR             Address of expansion dictionary
         ST    3,CMPSC_DICTADDR        Set dictionary address
         L     3,EXPADDR
         ST    3,CMPSC_SOURCEADDR      Set compression area
         L     3,EXPLEN
         ST    3,CMPSC_SOURCELEN       Set compression length
         LA    3,WORKAREA
         ST    3,CMPSC_WORKAREAADDR    Set work area address
MORE     DS    0H                      Label to continue
*
* Your code to allocate an 80-byte output area would go here
*
         ST   x,CMPSC_TARGETADDR      Save target expansion area
         LA   3,80                    Set its length
         ST   3,CMPSC_TARGETLEN       Set expansion length
         CSRCMPSC CBLOCK=CMPSC        Expand
         C    15,=AL4(CMPSC_RETCODE_TARGET) Not done, target used up
         BE   MORE                    Continue with operation
         DROP 2
 .
 .
         DS   0F        Align parameter on word boundary
MYCBLOCK DS   (CMPSC_LEN)CL1    CBLOCK Parameter
EXPADDR  DS   A         Input expansion area
EXPLEN   DS   F         Length of expansion area
```

```
EDICTADDR DS  A         Address of expansion dictionary
          DS  0D        Doubleword align work area
WORKAREA DS   CL192     Work area
          CSRYCMPS ,    Get mapping and equates
```

Note that this code loops while the operation is not complete, allocating a new 80-byte output record. It does not have to update the CMPSC_BITNUM, CMPSC_SOURCEADDR, or CMPSC_SOURCELEN fields, because the service sets them up for continuation of the original operation.

If running in AR mode, the example would also have set the CMPSC_TARGETALET and CMPSC_SOURCEALET fields. The XC instruction zeroed those fields as needed when running in primary ASC mode.

Determining if the CSRCMPSC Macro Can Be Issued on a System

Do the following to tell if the system contains the software or hardware to run a CSRCMPSC macro:

1. **Determine if CSRCMPSC is available**, by running the following:

```
          L    15,16                   Get CVT address
          USING CVT,15                 Set up addressability to the CVT
          TM CVTFLAG2,CVTCMPSC         Is CSRCMPSC available?
          BZ NO_CSRCMPSC               Branch if not available
* Compression feature is available
  .
  .
NO_CSRCMPSC DS 0H
```

2. **Determine if the CMPSC hardware instruction is available**, by running the following:

```
          L    15,16                   Get CVT address
          USING CVT,15                 Set up addressability to the CVT
          TM CVTFLAG2,CVTCMPSH         Is CMPSC hardware available?
          BZ NO_CMPSC_HARDWARE         Branch if not available
* CMPSC hardware is available
  .
  .
NO_CMPSC_HARDWARE DS 0H
```

Compression/Expansion Examples

Example 1

Operation

Compress a data area. Note that the expansion dictionary must immediately follow the compression dictionary, and both must be aligned on page boundaries.

```
          LA    2,MYCBLOCK              Get address of parm
          USING CMPSC,2
          XC    CMPSC(CMPSC_LEN),CMPSC  Clear block
          OI    CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_5 Set size
*                      Symbol size is 5+8. Dictionary has
*                      2**(5+8) entries
          L     3,DICTADDR
          ST    3,CMPSC_DICTADDR        Set dictionary address
          L     3,COMPADDR
          ST    3,CMPSC_TARGETADDR      Set compression area
          L     3,COMPLEN
          ST    3,CMPSC_TARGETLEN       Set compression length
          L     3,EXPADDR
          ST    3,CMPSC_SOURCEADDR      Set expansion area
          L     3,EXPLEN
          ST    3,CMPSC_SOURCELEN       Set expansion length
          LA    3,WORKAREA
          ST    3,CMPSC_WORKAREAADDR    Set work area address
          CSRCMPSC CBLOCK=CMPSC
```

```
         DROP  2
 .
 .
         DS    0F        Align parameter on word boundary
MYCBLOCK DS    (CMPSC_LEN)CL1    CBLOCK parameter
COMPADDR DS    A         Output compression area
COMPLEN  DS    F         Length of compression area
EXPADDR  DS    A         Input expansion area
EXPLEN   DS    F         Length of expansion area
DICTADDR DS    A         Address of compression dictionary
         DS    0D        Doubleword align work area
WORKAREA DS    CL192     Work area
         CSRYCMPS ,
```

Example 2

Operation

Expand a data area. Note that the expansion dictionary must be aligned on a page boundary.

```
         LA    2,MYCBLOCK              Get address of parm
         USING CMPSC,2
         XC    CMPSC(CMPSC_LEN),CMPSC   Clear block
         OI    CMPSC_FLAGS_BYTE2,CMPSC_SYMSIZE_5 Set size
*                      Symbol size is 5+8. Dictionary has
*                      2**(5+8) entries
         OI    CMPSC_FLAGS_BYTE2,CMPSC_EXPAND  Do expansion
         L     3,EDICTADDR
         ST    3,CMPSC_DICTADDR         Set dictionary address
         L     3,EXPADDR
         ST    3,CMPSC_TARGETADDR       Set expansion area
         L     3,EXPLEN
         ST    3,CMPSC_TARGETLEN        Set expansion length
         L     3,COMPADDR
         ST    3,CMPSC_SOURCEADDR       Set compression area
         L     3,COMPLEN
         ST    3,CMPSC_SOURCELEN        Set compression length
         LA    3,WORKAREA
         ST    3,CMPSC_WORKAREAADDR     Set work area address
         CSRCMPSC CBLOCK=CMPSC
         DROP  2
 .
 .
         DS    0F        Align parameter on word boundary
MYCBLOCK DS    (CMPSC_LEN)CL1    CBLOCK Parameter
EXPADDR  DS    A         Output expansion area
EXPLEN   DS    F         Length of expansion area
COMPADDR DS    A         Input compression area
COMPLEN  DS    F         Length of compression area
EDICTADDR DS   A         Address of expansion dictionary
         DS    0D        Doubleword align work area
WORKAREA DS    CL192     Work area
         CSRYCMPS ,
```

Example 3

Operation.

When using register notation in the CBLOCK parameter, the program must place both the address and ALET into a GPR/AR pair. This is true whether you are running in AR or primary ASC mode.

```
 .
 .
         LAE   2,MYCBLOCK              Set address *and* ALET
         CSRCMPSC CBLOCK=(2)           Issue operation
 .
 .
```

Appendix A. Assemble and Link-Edit Programs Using IOCS

You may assemble your declarative (DTFxx) macros, and any logic modules which you code yourself, in one of the ways as summarized below.

- With your main program.

  If you assemble your DTFxx macros and logic modules with the main program, the linkage editor searches the input stream and resolves the symbolic linkages between the DTF tables and modules.

- Separately for later link-editing with the main program.

  This requires that you specify the operand SEPASMB=YES in the DTFxx macro or xxMOD macro which is to be assembled separately. For the DTFxx macro, there are some symbolic linkages which you must define in your program in the form of EXTRN and ENTRY symbols.

  The SEPASMB=YES operand in a DTFxx macro causes:

  1. A CATALOG command with the specified file name to be punched out ahead of the object deck.
  2. This name to be defined as an ENTRY point in the assembled module.

  In an xxMOD macro, this operand causes a similar action: a CATALOG command with the module name to be punched out ahead of the object deck and this module name to be defined as an ENTRY point in the module.

  In either case, a START card must not be used.

This appendix discusses how you can make good use of separately assembled and cataloged DTFxx and xxMOD macros. It shows and discusses an IOCS sample program.

Cataloging Assembled DTFxx and xxMOD Macros

Much coding effort is saved if reusable DTFxx and xxMOD macros are cataloged in a sublibrary after they have been assembled. To use a cataloged DTFxx macro, use its name in all references your program makes to that file definition. If you name an xxMOD yourself (instead of letting IOCS do it), make sure that you refer to precisely that module in the related DTFxx macro. The linkage editor can perform an autolink only if there is a match of the module name specified in the DTFxx macro and the name of the module itself.

If a standard set of logic modules was generated when your VSE system was installed, auto-linking suitable modules to assembled DTFxx macros presents no problem. This is particularly true if both the modules and the DTF-block references use standard module names.

Assembling and Cataloging IOCS Modules

This requirement varies from installation to installation and depends on the kind of application programs used. It may arise if your migration to a new release involves also the installation of a new I/O device other than disk or tape.

Perform this step only if the release you get:

1. Includes the support of a new unit-record device (a printer or optical mark reader, for example).
2. Your system's configuration includes this new device.
3. Your system does not include a compatible I/O module.

IBM supplies IOCS (Input/Output Control System) modules needed by the z/VSE supported compilers. Since these modules may be linked also to user-written programs, there may be no need for you to

catalog IOCS modules of your own. For a list of IOCS modules shipped with z/VSE, see the *Program Directory*.

Figure 63 on page 182 shows a job stream example for assembling and cataloging IOCS modules for the **IBM 3525** punch device.

```
* $$ JOB JNM=IOMOD,DISP=D,PRI=8,                                      C
* $$ NTFY=YES,                                                        C
* $$ LDEST=*,                                                         C
* $$ CLASS=0
// JOB IOMOD
* **********************
* *
* * THIS JOB CATALOGS I/O MOD TO IJSYSRS.SYSLIB
* *
* **********************
// LIBDEF *,SEARCH=(PRD1.MACLIB,PRD2.GEN1)
* $$ PUN MEM=IOMOD.CATAL,S=IJSYSRS.SYSLIB,PUN=FED,REPLACE=YES
// OPTION DECK
// EXEC ASMA90,PARM='SIZE(MAX,ABOVE)'
             PRINT NOGEN
             CDMOD CTLCHR=ASA,IOAREA2=YES,DEVICE=3525,TYPEFLE=OUTPUT,  C
               SEPASMB=YES
             END
/*
* $$ PUN PUN=FED
// EXEC LIBR,PARM='MSHP'
AC S=IJSYSRS.SYSLIB
* $$ SLI MEM=IOMOD.CATAL,S=IJSYSRS.SYSLIB
/*
/&
* $$ EOJ
```

Figure 63. Job Stream for Assembling and Cataloging IOCS Modules

Note: The statement

```
// EXEC ASMA90,PARM='SIZE(MAX,ABOVE)'
```

calls the High Level Assembler. Refer to the topic that describes "Migrating From Earlier Releases" in z/VSE Planning, for details about calling the High Level Assembler.

IOCS Sample Program

The program used as an example and described in this section performs a card-to-disk operation with the following equipment and options:

- Card reader (SYS004).
- Disk (with user labels supplied).
- Record size: 80 bytes.
- Block size: 408 bytes – five logical records and an eight-byte count field per block.
- One I/O area and work area for the card reader.
- Two I/O areas for the disk.

The following methods can be used to provide the DTF blocks and logic modules for the program:

1. DTFxx macros, IOCS logic modules, and your program assembled together.
2. Logic modules assembled separately.
3. DTFxx macros and logic modules assembled separately.

   You can do this with the I/O areas defined with the DTFxx macros or within your main program. You can code your exit routines for the processing of labels or for error handling as part of the DTFxx macros or as part of the main program.

Figure 64 on page 183, a skeleton sample program, shows how the source code is to be submitted for separate assembly.

After having assembled and link-edited the program, it relates to the DTF tables and logic modules as shown in Figure 65 on page 184. The disk logic module is a phase supplied by IBM residing in the shared virtual area.

The examples are followed by a comparison of the three methods mentioned above.

```
                                                                  Column 72
CDTODISK  START   0
          BALR    12,0              INITIALIZE BASE REGISTER
          USING   *,12              ESTABLISH ADDRESSABILITY
          LA      13,SAVEAREA       USE REGISTER 13 AS A POINTER TO
*                                   THE SAVE AREA
          OPEN    CARDS,DISK        OPEN BOTH FILES
NEXT      GET     CARDS,(2)         READ ONE CARD AND HAVE IT MOVED
*                                   TO THE DISK OUTPUT BUFFER
*
          PUT     DISK              INCREMENT REGISTER 2 TO NEXT RE-
*                                   CORD LOCATION IN THE DISK BUFFER
          B       NEXT              RETURN FOR NEXT CARD
SAVEAREA  DS      9D                72-BYTE SAVE AREA, DOUBLEWORD
*                                   ALIGNED
EOFCD     CLOSE   CARDS,DISK        ON EOF OF THE CARD FILE, CLOSE
          EOJ                       BOTH FILES AND EXIT TO JOB CONTROL
MYLABELS  EQU     *                 YOUR LABEL-PROCESSING ROUTINE
           ...
           ...             The user-written label-processing routine
           ...
          LBRET   2                 RETURN TO MAIN PROGRAM
*   DEFINITION FOR THE CARD FILE
CARDS     DTFCD   DEVADDR=SYS004,                                       X
                  EOFADDR=EOFCD,                                        X
                  IOAREA1=A1,                                           X
                  WORKA=YES
DISK      DTFSD   BLKSIZE=408,                                          X
                  IOAREA1=A2,                                           X
                  IOAREA2=A3,                                           X
                  IOREG=(2),                                            X
                  LABADDR=MYLABELS,                                     X
                  RECFORM=FIXBLK,                                       X
                  RECSIZE=80,                                           X
                  TYPEFLE=OUTPUT
A1        DS      80C               CARD-INPUT BUFFER
A2        DS      408C              FIRST DISK BUFFER
A3        DS      408C              SECOND DISK BUFFER
          CDMOD   DEVICE=2540,      CARD I/O LOGIC MODULE               X
                  TYPEFLE=INPUT,                                        X
                  WORKA=YES
          END     CDTODISK          PROGRAM-START ADDRESS
```

Figure 64. IOCS Sample Program - Source Code for Common Assembly

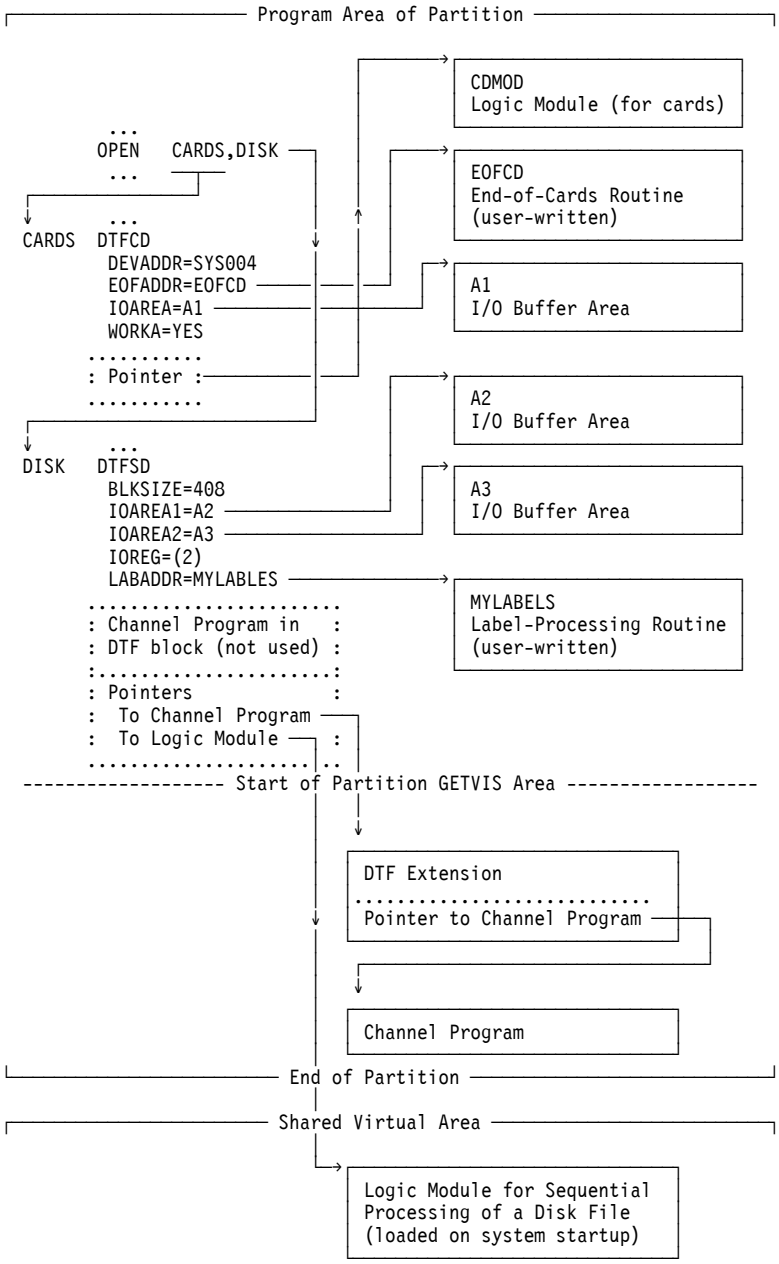


Figure 65. Program, DTF Tables, and Logic Modules in Storage

Assemble the DTFs and Logic Modules Separately

Some minor modifications to the program are necessary as shown in Figure 66 on page 185.

The Main Program

```
     CDTODISK  START   0
               BALR    12,0
               USING   *,12
               LA      13,SAVEAREA
               OPEN    CARDS,DISK
     NEXT      GET     CARDS,(2)
               PUT     DISK
               B       NEXT
     SAVEAREA  DS      9D
(1)            EXTRN   CARDS,DISK
               END     CDTODISK
```

The Card-File-Definition Macro (DTFCD)

```
                                        Column 72
     CARDS     DTFCD                        X
                       DEVADDR=SYS004,      X
                       SEPASMB=YES,         X
                       EOFADDR=EOFCD,       X
                       IOAREA1=A1,          X
                       WORKA=YES
               USING   *,14
     EOFCD     CLOSE   CARDS,DISK
               EOJ
(1)            EXTRN   DISK
     A1        DS      80C
               END
```

(1)

If the main program is assembled separately from the DTFxx macros, the assembler cannot resolve the references to CARDS and DISK. Therefore, these names must be defined as external symbols.

The Disk File Definition Macro (DTFSD)

```
                                        Column 72
     DISK      DTFSD    BLKSIZE=408,        X
                        SEPASMB=YES,        X
                        IOAREA1=A2,         X
                        IOAREA2=A3,         X
                        IOREG=(2),          X
                        LABADDR=MYLABELS,   X
                        RECFORM=FIXBLK,     X
                        RECSIZE=80,         X
                        DEVICE=3350,        X
                        TYPEFLE=OUTPUT
     MYLABELS  BALR    10,0
               USING   *,10
               ...
               LBRET   2
     A2        DS      408C
     A3        DS      408C
               END
```

The Logic-Module Generation Macro (CDMOD)

```
     CDMOD     DEVICE=2540,                X
               TYPEFLE=INPUT,              X
               WORKA=YES                   X
               SEPASMB=YES
     END
```

Figure 66. Source Code Modifications for a Separate Assembly

After the assembly of these source decks, each DTFxx macro and also the logic module for card I/O is preceded by the CATALOG command. This command is needed to catalog the macros and the module as

library members of type OBJ. To run the entire program, all modules must be processed by the linkage editor. After the link-edit run, your program relates to the DTF tables and logic modules in the same way as shown in Figure 65 on page 184.

As mentioned earlier, you must do some minor modifications to your source code. The kind of modifications to be done is discussed with reference to Figure 66 on page 185.

1. In the assemblies of the DTFCD and DTFSD macros, a USING statement was added because certain routines were separated from the main program and moved into the DTF assembly.

   When a routine (such as error or label-processing) is separated from the main program, addressability must be ensured for the routine. You can provide this addressability by assigning and initializing a base register. Only for an end-of-file (EOF) routine is the addressability established by IOCS (by loading an address into register 14).
2. Any reference to a name that is not included in the code submitted to the assembler must be defined as an external symbol by way of an EXTRN assembler statement. In this example, the following names must be defined as external:
   - In the main program – The names of the separately assembled DTFxx macros: CARDS and DISK.
   - In the DTFCD macro – The name of the separately assembled DTFSD macro: DISK.
   - In the DTFSD macro – None.

The object decks can be cataloged into a sublibrary. The job shown below catalogs them into the sublibrary that you defined as the TO sublibrary in a job control LIBDEF statement.

```
// JOB CATLIB
// EXEC LIBR
   ACCESS  libname.sublibname
    ... (DTFCD Assembly)
    ... (DTFSD Assembly)
    ... (CDMOD Assembly)
/*
/&
```

For libname.sublibname, supply the name of the sublibrary (qualified by the name of its library) into which the object decks are to be cataloged.

Alternatively, the object decks from these assemblies (DTF tables and logic modules) can be submitted to the linkage editor along with the object deck of the main program. The job for this is:

```
// JOB CATALOG
// LIBDEF CATALOG=libname.sublibname
// OPTION CATAL
   INCLUDE
   PHASE name,*
    ... (object deck, main program)
    ... (object deck, DTFCD assembly)
    ... (object deck, DTFSD assembly)
    ... (object deck, CDMOD assembly)
/*
// EXEC LNKEDT
/&
```

For more information about the required control statements, see z/VSE System Control Statements.

Comparison of the Three Possible Methods

Assemble the Program, DTFxx Macros, and Logic Modules Together

It requires the most assembly time and the least link-edit time. Because the linkage editor is substantially faster than the assembler, this method of frequent reassembly of a program takes more total time for program preparation than the other methods.

Assemble the Logic Modules Separately

This method separates the IOCS logic modules from the remainder of the program. Because these modules are generalized, they can serve several different applications. They are normally retained in a sublibrary for ease of access and program service. Consider the following to improve overall system availability:

1. Identify the IOCS logic modules that your application programs may need.
2. Generate these modules and catalog them in a sublibrary that is accessible from all partitions. Each of these modules requires an assembly and a catalog operation. However, the assemblies can be batched together as can be the catalog jobs.

Object programs produced by COBOL, PL/I, and RPG require one or more IOCS logic modules in each executable program. These modules are usually assembled (and linked into a sublibrary) during system installation.

Assemble the DTFxx Macros and Logic Modules Separately

This method enables you to create a standardized IOCS package separated almost totally from a main program. Only the imperative IOCS macros (such as OPEN, CLOSE, GET, and PUT) remain. All file definitions, label processing and other IOCS exits, and possibly also the buffer areas are preassembled.

If there are few IOCS changes in an application, compared to other changes, this method significantly reduces the time needed for program development and program service. The method also helps to standardize file descriptions so that they can be shared among several different applications. It reduces the chance of one program creating a file that is improperly accessed by subsequent programs.

When using this method, you need be concerned only with the record format and the general register pointing to a record. You can virtually ignore operands such as BLKSIZE and LABADDR in your program, although you must ultimately consider their effect on virtual storage, job control statements, and so on.

If you build output records in an I/O area, you might want to define this area in your main program rather than with the associated DTFxx macro. Likewise, the label-processing and other exit functions can be moved into the main program. If an exit function is to be standard across all of your applications, assemble it together with the associated DTFxx macro. If each application requires a special exit function, assemble it into the main program.

Appendix B. Direct Access Method (DAM)

DAM, a flexible access method, is available for use with **CKD and ECKD disk** devices.

With this access method, you can process the records of a file in random order. The records may be read, written, updated or replaced.

DAM supports unblocked records of any format that is supported by VSE. When record spanning is used, the segmentation of the logical records and their reassembly is performed by DAM routines whenever necessary.

DAM uses one I/O area for a file. To determine the size of this area, take the following into account:

- The length of the data area
- Your program's use of the count and key areas
- The control information

To process records in random order, your program must make the address of the record available to DAM and issue a READ or WRITE macro to have DAM transfer the record.

A file to be read or written must be defined with the DTFDA macro. A logic module for the file need not be coded. To understand the purpose of some of the operands of the DTFDA macro, you may need information about how DAM makes use of these operands.

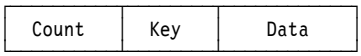
Defining the File

This section discusses the things you should consider when you code the DTFDA macro defining your file.

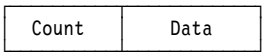
Record Types

Records to be processed by DAM can be stored on disk in either of the formats shown below:

Format with key area:



Format without key area:



If your program processes spanned records, this format applies only to the first segment of each record.

If the records of a file have keys that are to be processed, every record must have a key and all keys must have the same length. If you do not specify the KEYLEN operand in your DTFDA macro, then:

- DAM ignores keys, if present.
- You cannot use a formatting WRITE macro (WRITE AFTER) to extend the file.
- For a WRITE ID or a READ ID, DAM writes or reads only the data portion of the record.

DAM considers all records as unblocked. Provide for blocking and deblocking in your program, if blocking of records on disk is desirable.

The records of your file can be of fixed or variable length, or they can be undefined. The type of records in the file must be specified in the RECFORM operand of your DTFDA macro.

A spanned record format indicates blocks of variable length, where the size of each segment is a function of the track size and record size. The record size is set by a WRITE AFTER macro. All the variable record segments of a given spanned record are logically contiguous.

To write records that are specified as undefined, your program must, for each record:

1. Determine the length of the record.
2. Load this length into the register specified by the RECSIZE operand of the DTFDA macro.
3. Issue a WRITE macro to have DAM write the record.

I/O Area Specification

The DTFDA IOAREA1 operand defines an area of virtual storage into which records are read on input or in which records are built on output. The length of this area is specified in the BLKSIZE operand.

The format of the I/O area depends on your specifications in the DTFDA macro. The section I/O Area for a DAM File with Fixed Unblocked and Undefined Records shows:

- which operands of the DTFDA macro affect the format of the area when the records of your file are of fixed length and unblocked
- which operands of the DTFDA macro affect the format of the area when the records of your file are of an undefined format
- the format of the area for unblocked records of variable-length
- the format of the area for unblocked spanned records

Use these illustrations to determine the length of the I/O area for your file. The area must be large enough to include an eight-byte count field, if necessary. The count itself is provided by DAM.

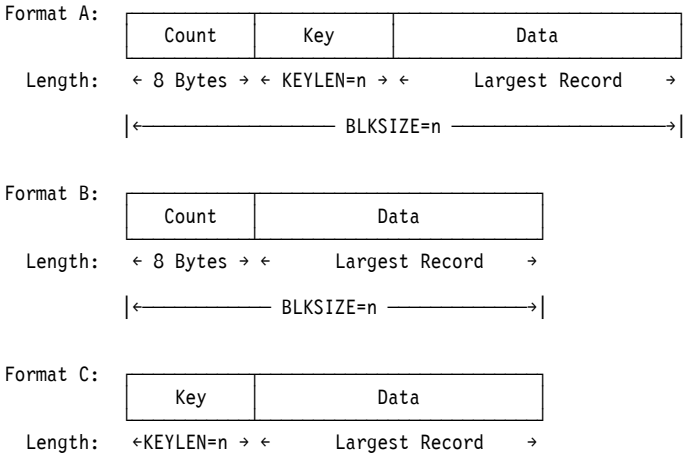
I/O Area for a DAM File with Fixed Unblocked and Undefined Records

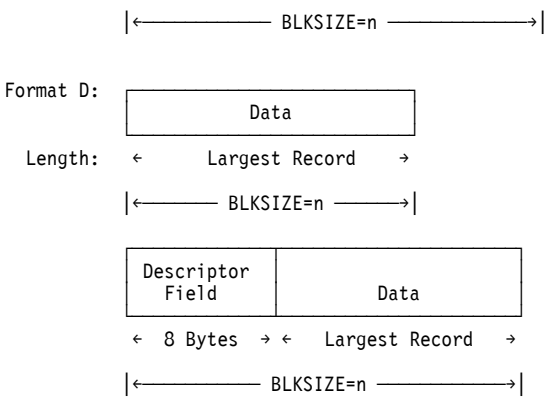
E =
Either or both of the two operands so marked.

O =
Optionally with the operand.

X =
Specified to meet your requirements

Operand of DTFDA						Format of I/O Area
AFTER	KEYLEN	READID	WRITEID	READKEY	WRITEKY	
X	X	O	O	O	O	Format A
X		O	O			Format B
	X	E	E	O	O	Format C
	X			E	E	Format D
		E	E			Format D





When you build a record, you must supply, in the subareas as shown, all of the various information types, except the count information. For example, if your DTFDA defines a file to which Format A applies, the data of a record would be located to the right of the key area and the key area to the right of the count area.

For variable-length and spanned unblocked records, the format of the I/O area is independent of the DTFDA macro operands. If you specify the KEYLEN operand of the macro, DAM passes the key to or takes it from the field you specified in the KEYARG operand. The count field, if desired, is taken from a field put aside by DAM internally.

The descriptor field is built by DAM, except when you create your file or add records to it by using the WRITE AFTER macro. In that case you must insert, into bytes 4 and 5 of this field, the sum of the data length of the record plus 4. If you read a variable length or spanned unblocked record, these bytes indicate the length of the record. If you update records, do not change any byte of the field.

Following are the maximum lengths for logical records on the various types of disk devices:

	RECFORM=	
Device	FIXUNB, VARUNB, UNDEF	SPNUNB
3350	19␣069	32␣767
3375	32␣767	32␣767
3380	32␣767	32␣767
3390	32␣767	32␣767
9345	32␣767	32␣767

Logic Modules

A VSE system includes preassembled and linked superset logic modules. The system loads these modules into the SVA during system startup. DAM links a suitable module to your DTFDA table when the file is opened. These logic modules are fully reentrant. Thus one copy of a logic module can be used by all requesters having the type of file for which the logic module was generated.

Processing the File

Following is a list of imperative macros that you can use to process a file defined by a DTFDA macro:

- CLOSE(R)
- LBRET
- CNTRL
- OPEN(R)
- ERET
- READ
- FEOVD
- WAITF
- FREE

- WRITE

After a DAM file has been defined (by a DTFDA macro), you use the imperative macros to operate on the file. These macros are divided into three groups: to open the file, to process the file, and to close the file.

Opening the File

The OPEN macro must be used to activate a DAM file for processing. The macro associates the logical file defined in your program with a file of data on disk. This association remains in effect until you issue a CLOSE macro for the file.

When OPEN attempts to activate a file whose device is unassigned, DAM cancels the job. If the device is assigned IGN, DAM opens but does not activate the file; instead, DAM returns control to your program. In your program, do not request any I/O for this inactive file. DAM indicates an inactive, open file by setting on bit 2 of the DTF byte 16.

If you plan the processing of standard user labels (UHL only), your program must provide the information for checking or building the labels. If this information is obtained from another input file, that other file must be opened first. Specify that other input file ahead of your DTFDA disk file in the same OPEN or issue a separate OPEN for the input file preceding the OPEN for the DTFDA file.

If the XTNTXIT operand is specified, OPEN stores the address of a 14-byte extent-information area in register 1 and gives control to your extent routine. You can save this information for use in specifying record addresses. Then the next volume of the file is opened (on an input file, only after the user labels, if any, have been processed). When all volumes are open, the file is ready for processing. If the disk device is file-protected, all extents specified in EXTENT statements are available for use.

DAM requires that all volumes of a multivolume file are mounted and opened before processing can begin.

For each volume of an **output** file, DAM checks the standard VOL1 label and the extents specified; it checks the EXTENT statements for the following:

- The extents are of type 1.
- The extents do not overlap.

  Open checks all the labels in the VTOC to ensure that the new file does not write over an existing, unexpired file. Open then builds the standard label(s) for the file and writes the label(s) into the VTOC.
- The first extent is at least two tracks long if user standard header labels are to be written.

  OPEN reserves the first track of the first extent for these header labels and gives control to your label routine. Use the LBRET macro in your routine to return control to DAM.

For each volume of an **input** file, Open checks the standard VOL1 label and the file label(s) in the VTOC. Open also checks some of the information specified in the EXTENT statements for that volume.

If LABADDR is specified, OPEN makes the user standard labels available to your routine, one after the other, one at a time for checking. Use the LBRET macro in your routine to return control to DAM.

Use OPENR instead of OPEN if your program is self-relocating. This causes DAM to relocate all address constants within the DTF table of each file that is to be opened. However, a zero constant is relocated only when it represents the module address.

Creating a File and Adding of Records to a File

Your program can preformat a file or an extension of a file in one of two ways. The method you choose depends on the type of processing to be done:

- If you use only the WRITE AFTER macro in your program, the WRITE RZERO macro preformats the tracks.
- If you use nonformatting functions of the WRITE macro in your program, preformat the tracks by performing a Clear Disk utility run. This utility resets the capacity record to reflect an empty track. How to use this utility is described in z/VSE System Utilities.

When you create a file or write new records into a DAM file, each of your records is written with the count area, the key area (if present), and the data area. DAM writes the new record behind the one written last on the specified track. The remainder of the track is erased. You specify this method in a WRITE macro by the operand AFTER.

DAM ensures that each record fits on the specified track before it writes the record. If a record does not fit, DAM sets a no-room-found indication in your error/status byte specified by the ERRBYTE operand of the DTFDA macro for your file. If WRITE AFTER is specified, DAM determines, from the capacity record, the location where the record is to be written.

If the DTFDA macro for your file specifies the AFTER option, DAM uses the first record on each track (R0) to maintain information about the data records on the track. The layout of this record (see Figure 67 on page 193) may be of interest for analyzing the dump of a certain disk area.

Bytes
Contents

0-7
Count Area

Bytes

0
Used by DAM.

1-5
Identifier: cchhr of the data area.

6-7
Used by DAM.

8-15
Data Area

Bytes

8-12
Identifier (cchhr) of the last record on the track.

13-14
Unused-bytes count: the number of bytes remaining

:
on the track.

15
Reserved.

Figure 67. Contents of Record 0 with Capacity-Record

Locating a Record

DAM requires two references for all read or write operations: the track reference and the record reference:

- The track reference can be either of the following:
  - The actual disk address, which specifies the location of the track.
  - The relative track address, which specifies the position of the track relative to the beginning of the file.
- The record reference can be either the record key (if the records contain key areas) or the record identifier (ID).

DAM seeks the specified track, searches it for the desired record location, and reads or writes the record as indicated by the macro. If it cannot find a record, DAM indicates this in the error/status bytes as specified by the ERRBYTE operand of the DTFDA macro for your file (for more information about these two

bytes, see the section “Error Handling” on page 207). Have your program test this indication and take a suitable action.

Multiple tracks can be searched for a record specified by key (SRCHM). However, if it cannot find a record after having searched an entire cylinder (or the remainder of a cylinder), DAM sets an end-of-cylinder rather than a no-record-found indication.

When your program issues an I/O request, DAM returns control to your program immediately. Therefore, when your program is ready to process the input record, or build the next output record for the same file, your program must ensure that the previous transfer of data is complete. You ensure this by coding a WAITF macro in your program.

After having completed a READ or WRITE request for a certain record, DAM can make the ID of the next record available to your program. You must set up a field (in which to store the ID) if you request DAM to supply the ID. Specify the symbolic address of this field in the IDLOC operand of the DTFDA macro for your file. When the record reference is by key and multiple tracks are searched, DAM supplies the ID of the specified record rather than the next record. This can be useful for a random update operation, or for the processing of successive records. If your program processes records consecutively, based on the next ID, and the file does not have an end-of-file record, the program can check the ID supplied by DAM against the end-of-file limit.

Track Reference

To provide a track reference, do the following:

- Set up a track reference field in your program.
- Specify the name you assigned to this field in the SEEKADR operand of the DTFDA macro for your file.
- Determine, by specifying the applicable DTFDA operands, the type of addressing to to be used.

Before issuing any READ or WRITE macro for a record, store the identifier of the track that is to be accessed in the track reference field. You do this in one of the following ways:

- In the first seven bytes – *mbbcchh* in hexadecimal format.
- In the first three bytes – *ttt* in hexadecimal format.
- In the first eight bytes – *tttttttt* in zoned decimal format.

The latter two (track) references, along with the operands DSKXTNT RELTYPE, indicate that relative addressing is to be performed. Thus, instead of providing the exact physical track location (mbbcch), only the track number relative to the starting track of the file need be provided. If these operands are omitted, the physical track location is assumed. When it executes the READ or WRITE macro, DAM refers to the track reference field to select the desired track.

Addressing for direct access of data on disk may be different if you use a high-level programming language, such as COBOL or PL/I. For information about addressing data in a high-level programming language, consult the applicable language reference publication.

Physical Track Addressing

This kind of addressing is used as a starting point for a search on record key (control field) or as the actual address for a read or write operation. To request a search for a key, specify that this search is to be one of the following:

- Only within the specified track (hh).
- From track to track starting at the given address until the end of the cylinder (cc) is reached.
- From track to track starting at the given address until the record is found.

For details about setting up a track-reference field for physical track addressing, see Figure 68 on page 195.

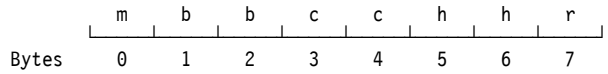


Figure 68. Track Reference Fields – Physical Track Addressing

where:

m

Identifies the volume. If a single file extends over more than one volume, the physical units must be assigned (by job control EXTENT statements) to a sequential set of logical unit names. The value of m is always 0 for the first volume, 1 for the second, 2 for the third, and so on. For example: a file spread over three volumes could be assigned to the logical units SYS002 through SYS004. In this case:

- m = 0 refers to SYS002
- m = 1 refers to SYS003
- m = 2 refers to SYS004

bb

Reserved.

cc

A two-byte field that contains the number of the cylinder in which the record can be located. This cannot be the number of a cylinder reserved for alternate tracks. The two bytes, together with the bytes hh, provide the track identification.

DAM does not allow you to use different data module sizes for a multi-volume file on CKD devices.
(If mini-disks are defined for
the volumes, these mini-disks must have the same number of cylinders.)

The number is in binary format.

hh

Contains, in binary format, the head number in the second byte. The first byte is reserved.

r

The record number within a track. This one-byte field can contain a binary value of 0 - 255 to identify the location of a record on the track. This field is not used when records are referenced by record key.

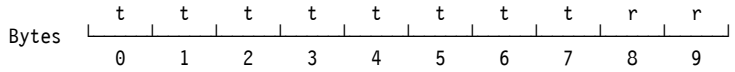
Relative Track Addressing

The required disk address may be given as a relative address (as shown in Figure 69 on page 196). This address is then converted by DAM to an actual address. Relative track addressing is more convenient to use than the physical address because:

- Your file is treated logically as if it were located in one continuous area, although it may occupy several nonadjacent areas.
- You need to know only the relative position of the data within the file; the actual address of this data is of no concern. This can be of advantage if you plan to move your file from one location to another. The relative addressing scheme remains the same, but the actual addresses may change.

How to set up a track-reference field for relative track addressing is shown in Figure 69 on page 196.

Decimal Identifier



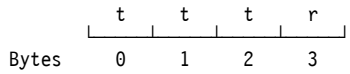
where:

tttttttt =
Track number relative to the beginning of the file.

rr =

Record number relative to the beginning of the track. If your program refers to a record by key, rr should be zero.

Hexadecimal Identifier



where:

ttt =
Track number relative to the beginning of the file.

r =

Record number relative to the beginning of the track. If your program refers to a record key, r should be zero.

Figure 69. Track Reference Fields – Relative Track Addressing

Format of the Record ID

For certain types of operations, you can request DAM to return the actual record address (ID) of the block read or written, or of the block following the one just read or written. Your program can use this returned ID to read or to write a new record, or to update the one just read and write the updated record back to the same location.

The format of the returned ID is the same as the format of the disk address used for locating the record, that is, one of following:

- *mbbcchhr* (physical track addressing)
- *tttttttrr* (relative track addressing, decimal identifier)
- *tttr* (relative track addressing, hexadecimal identifier)

Record Reference

DAM allows records to be specified either by record key or by record identifier.

Reference by Record Key

If the records of your file include keys, DAM can search them randomly by their keys. This allows you to refer to records by their logical control information such as an employee number, a part number, a customer number, and so on.

To refer to records by key, you must:

1. Set up a key field in your program.
2. Specify the name of that field in the KEYARG operand of the DTFDA macro for your file.
3. Store the key of a desired record in this field before you issue the request (READ or WRITE) macro.

Reference by Identifier (ID)

The records on a track can be searched randomly by their position on the track rather than by control information (key). To do this, use the record identifier.

- The physical record identifier

  This is part of the count area of the disk record; it consists of five bytes: cchhr. The first four bytes (cylinder and head) refer to the location of the track; the fifth byte (record) uniquely identifies the record on the track.

- The relative track notation

  To use this notation, DAM requires you to specify the operands DSKXTNT and RELTYPE in the DTFDA macro for your file. DAM requires that the records are numbered in succession (without missing numbers) on each track: 1 for the first record, 2 for the second record, and so on. The r-byte(s) of the track-reference field (see Figure 69 on page 196) must contain the number of the desired record. When processing a READ or WRITE macro that searches by ID, DAM refers to the track-reference field to determine which record is requested by the program. DAM compares the number in this field with the corresponding fields in the count areas of the disk records.

Locating a Free Space

DAM maintains a count of unused bytes in record zero on each track (as shown in Figure 67 on page 193). When a record is to be written, DAM uses this count to find out whether this record fits on the track. If the new record:

- Fits on the track, DAM writes the record as a new last record and updates the unused-bytes count.
- Does not fit on the track, DAM notifies your program. An overflow routine in your program may then become active.

Have your conversion algorithm randomize to a track address. If then a synonym results from your conversion algorithm, DAM writes the second and subsequent records with the same address behind the first one as long as there is room on the track. Thus, more than one record may have the same track address assigned.

The capacity record is not always used. The description of the WRITE macro in z/VSE System Macros Reference explains when the record is used.

DAM updates the capacity record for each data record that fills empty space on a track. However, when a record is deleted, the capacity record does not show it as empty space. Space that is freed because a record has been deleted can be recognized as such only by your program. It may, for example, do this:

1. After having deleted a record from a file of records with keys:
   a. Search on key to read the record and request DAM to return the record ID.
   b. Use this ID to write blanks or zeros (or whatever unique identification is acceptable to mark the deleted key and data area) to this location.
2. To re-use the deleted record for data:
   a. Randomize the key of a new logical record to a starting location.
   b. Make a search for a blank or zeroed key.
   c. Use the returned ID to write the new record with the new key into the same location.

Reading and Writing a Record

Once a DAM file has been opened, your program can use the imperative macros for direct access: READ, WRITE, WAITF, and CNTRL. This section discusses the macros together with the major processing functions of DAM.

Load and Process a File

The only difference between loading (creating) a direct access file and processing it (updating or retrieving records) is the file's initial status. In both cases, the same conversion algorithm is used for locating data records.

Before creating a file, you must ensure that the disk storage area is cleared of any data that may have been stored previously. IBM provides two programs that you can use to clear disk storage areas:

- Device Support Facilities

  The program operates on complete volumes. You use it for initializing a new volume or reinitializing a previously used volume.

  The program writes a preformatted VTOC and clears the entire volume. Afterwards, each track contains a home address and a record zero describing the entire track as free space. The preformatted VTOC contains empty file labels.

- Clear Disk

  This utility program operates on logical files. You use it to preformat a disk storage area with dummy blocks of fixed-length format. Preformatting by a Clear Disk run is a must for files with records of fixed length.

How your file is loaded and processed depends on whether your records have a key area.

Processing of Records with a Key Area

This processing varies for records of fixed length and records of variable length.

Fixed-Length Records with a Key

For the required clear-disk run, use the same fill character that you use to indicate a deleted record. That way you can use the same code for creating and for updating the file.

You can distinguish a current data record from a dummy record by the key of the record. The keys of a file's dummy records all have the same contents; current data records have unique keys, and each key identifies a certain data record.

You have two options for writing a record:

- Randomizing to a cylinder address.

  Specify the search-multiple-tracks option (SRCHM=YES in the DTFDA macro) and supply the address of the cylinder as produced by the randomizing algorithm of your program. This allows you to search for the first dummy record on a cylinder. The search starts at the beginning of that cylinder and continues until either a dummy record is found or the end of the cylinder is reached.

  When it finds a dummy record, DAM returns control to your program and passes the address of a record. You can write the new record at the address passed by DAM.

  If it does not find a dummy record, DAM indicates a no-record-found condition. Your overflow routine must then become active.

  The technique for locating a record slot in an independent overflow area is the same as that for the prime data area.

  Consider defining one or more cylinders as an independent overflow area. An overflow area of one or more tracks on each cylinder is useful only if these tracks are excluded by your randomizing algorithm.

- Randomizing to a track address.

  If you use the search-multiple-tracks (SRCHM=YES) option, the procedure is the same as above, except that the search begins at a specified track and not at the beginning of a cylinder. If you do not specify SRCHM=YES, the search for a dummy record ends at the end of the specified track.

  When DAM returns control, it passes to your program either:

– The address of the record following the one that DAM identified as a dummy record, or
– A no-record-found indication; in this case, you would normally branch to your overflow routine.

Searching for a specific track may be more time consuming, but it gives you more direct control.

You can choose between cylinder overflow areas and independent overflow areas. However, it is hard to say which method will be more efficient. If the prime data area and the independent overflow area reside on the same volume, switching to and from the overflow cylinders requires the read/write mechanism to move. You can avoid such movement by using cylinder overflow areas.

Variable-Length Records with a Key

Do not use the Clear Disk utility to preformat the required disk space. Use the Device Support Facilities program to clear a complete volume; to clear a certain area on a volume, write erased tracks and update the unused-bytes count in record 0 of each track (for a layout of a record 0, see Figure 67 on page 193). However, this count gives only the number of free bytes at the end of the track. Space that is free because a record has been deleted is not taken into account.

In your program, code a randomizing algorithm that returns a cylinder and a track address. DAM checks the unused-bytes count. If the record fits, DAM writes it behind the current last record on the track. If the record does not fit, DAM indicates this to your program. You can do a fit/no-fit inquiry in the overflow area in exactly the same way, track by track, until DAM finds a track on which the new record fits.

It may be useful to have both a cylinder and an independent overflow area. When a prime data track overflows, try first to store the record in the cylinder overflow area. If this is not possible, store the record in the independent overflow area.

A record stored in the cylinder overflow area can later be retrieved automatically if the search-multiple-tracks option is specified. To retrieve records that are stored in the independent overflow area, you must set up a search in your program. A record cannot be stored in the space occupied by deleted records. Therefore, a cylinder overflow area itself may overflow. To maintain processing efficiency, reorganization of the entire file may then soon be necessary. To reorganize your DAM file, do the following:

1. Read the file track after track.
2. Clear each track separately.
3. Restore each current data record as if it were new.

Since DAM does not restore deleted records, free space will concentrate again at the end of the tracks.

After the prime data tracks have been reorganized, the overflow area may be processed. DAM attempts to write overflow records to the prime data area. Overflow records that cannot be moved to the prime data area are moved back onto the overflow tracks. Again, deleted records are omitted.

Retrieving Records with a Key

A record can be retrieved by a search on key. If the option for a search on multiple tracks is specified, DAM can find a record on a cylinder if you specify the search to start at or before the address of the record. The conversion algorithm that is applied for writing a record can be used also for retrieving the record.

Processing of Records without a Key Area

If records are written without a key, the location of a record can be determined only by the randomizing algorithm. DAM can identify a desired record only by the disk address your program has supplied.

A practical method of randomizing is to code a conversion algorithm that computes a cylinder, track, and record address. To use this method, your file:

1. Must have fixed-length records.
2. The disk area it is to use must have been preformatted by a Clear-Disk run before the file is loaded.

Coding a conversion algorithm for data without a key is somewhat critical because each synonymous record becomes an overflow record.

In the prime data area, each block has a pointer field. As long as no synonym is present for a certain record in that area, this pointer is empty. If synonyms are present, this pointer points to the first synonym of a possible chain of overflow records. All synonyms for a record in the prime data area are linked by overflow chain pointers. This is shown in Figure 70 on page 200.

The chain of overflow pointers is used to trace a certain overflow record. It says nothing, however, about the actual location of the records on the track. The location of an overflow record is, in fact, of no concern to your program.

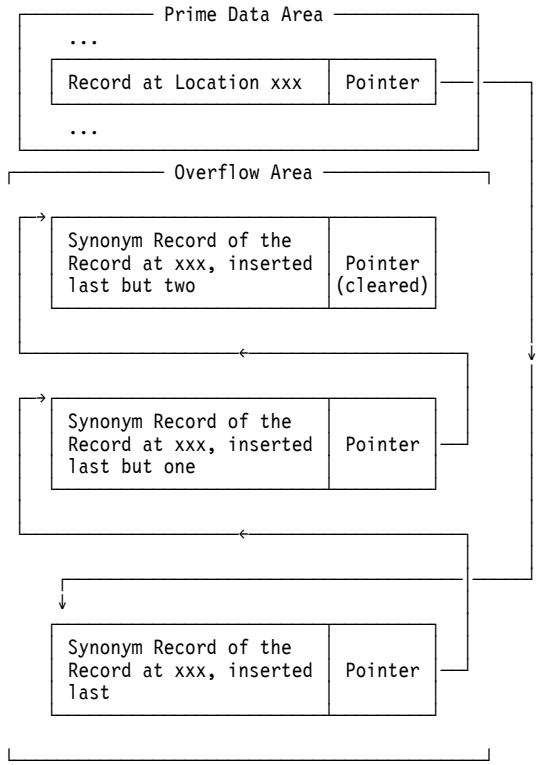


Figure 70. Prime Data Record and Related Overflow Records

In principal, your program must do the following:

1. Compute a DASD record address, using its randomizing algorithm.
2. Check whether the block at the computed address contains current data. This requires an input operation. If:
   - The block contains no current data.
     a. Clear the overflow pointer to make sure it indicates that no synonyms are present.
     b. Write the new record.
   - The block contains current data and the overflow pointer is empty.
     a. Have DAM find the address of a free record location for the synonym. How this can be done is described under “Getting the Address of a Free Record” on page 201.
     b. Put this address into the overflow pointer.
     c. Write the prime data block back to its original location on disk and write the new record as the first overflow record.
   - The block contains current data and the overflow pointer indicates the presence of synonyms.
     a. Save the address in the overflow pointer. This is the address of the first synonym in the overflow chain.
     b. Have DAM find the address of a free record location for the new synonym. How this can be done is described under “Getting the Address of a Free Record” on page 201.

c. Write this address into the overflow pointer.

d. Restore the prime data block to its original location on disk.

e. Put the former contents of the overflow pointer into the overflow chain pointer of the new synonym.

f. Write the new synonym at the address in the overflow area supplied by DAM.

As a result, the newly inserted record becomes the first synonym in the overflow chain, and the former first one becomes the second. The rest of the sequence remains unchanged.

Getting the Address of a Free Record

The randomizing algorithm calculates only prime data addresses. However, most likely you want to find the address of a free record location without having to search the overflow area block by block.

A good method is to use the first record of the area as an overflow-area descriptor. Have this record contain, at all times, the address of the first free block in the overflow area. If a new record is to be added to the overflow area, the descriptor gives the address of the block where the new record can be stored. When a new record is stored, the address in the descriptor is to be updated to point to the next free record.

The overflow-area descriptor record may contain any information you need in addition to a pointer to the first free block.

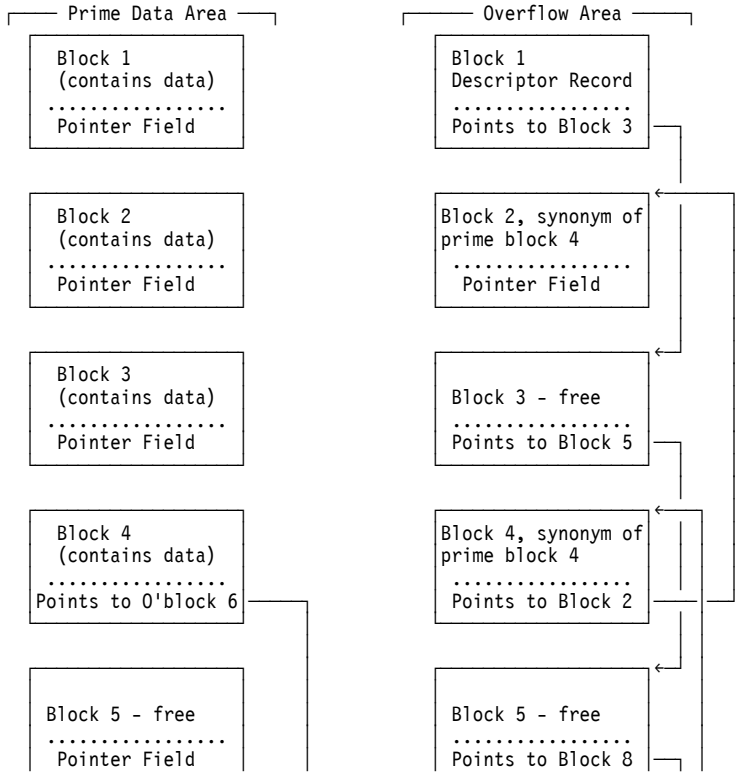
When a record is deleted from the overflow area, store the address of that block in the overflow-area descriptor record (this makes the space of the deleted record the first free record). In addition, store the address that was in the overflow-area descriptor record in the block that just became free. This gives you a pointer to the next free block.

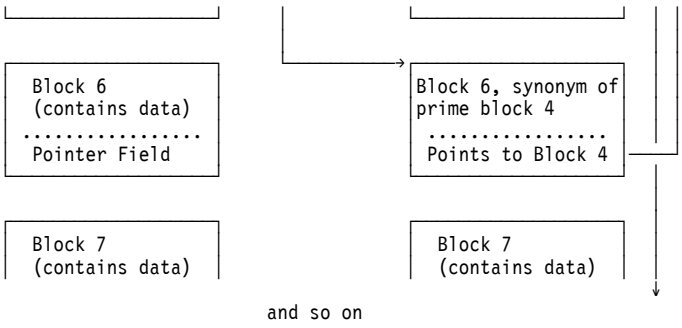
The examples below illustrate the process as described above.

```
———→   Existing pointers
.....→   Changed or new pointers
```

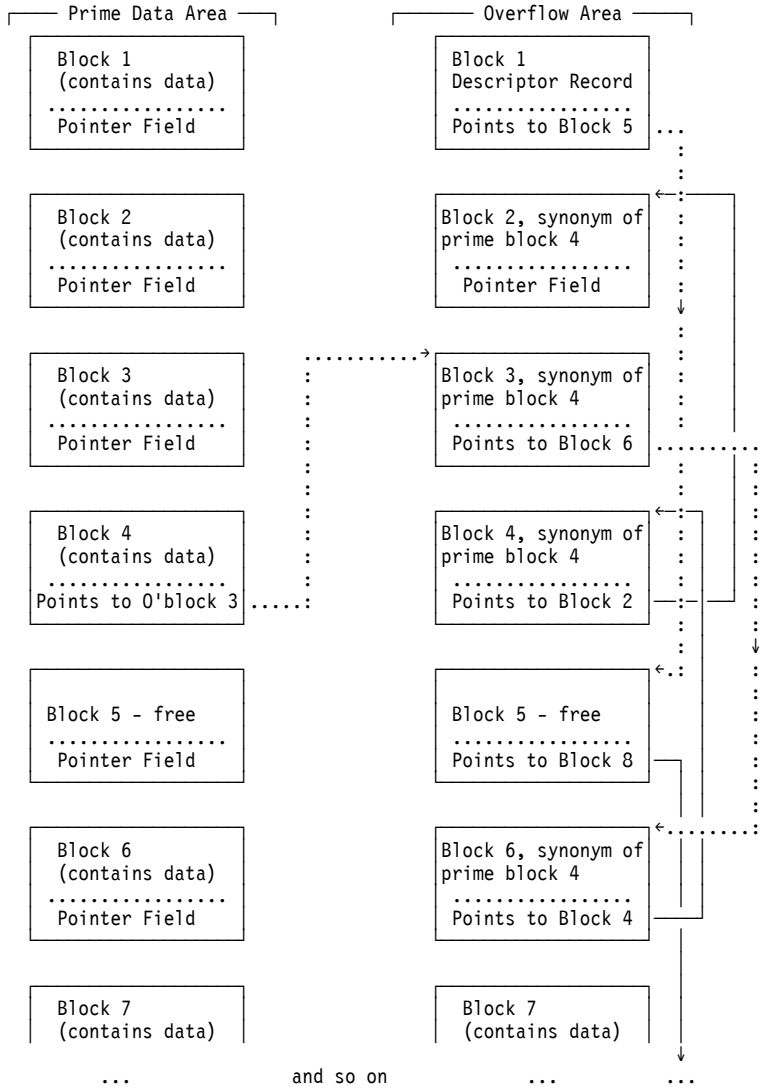
Sample Overflow Organization

Initial Status of the File

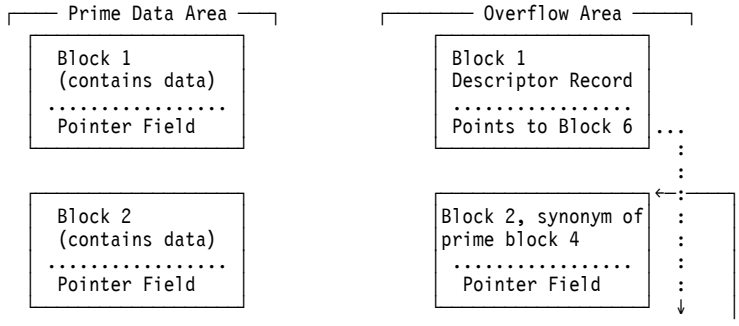


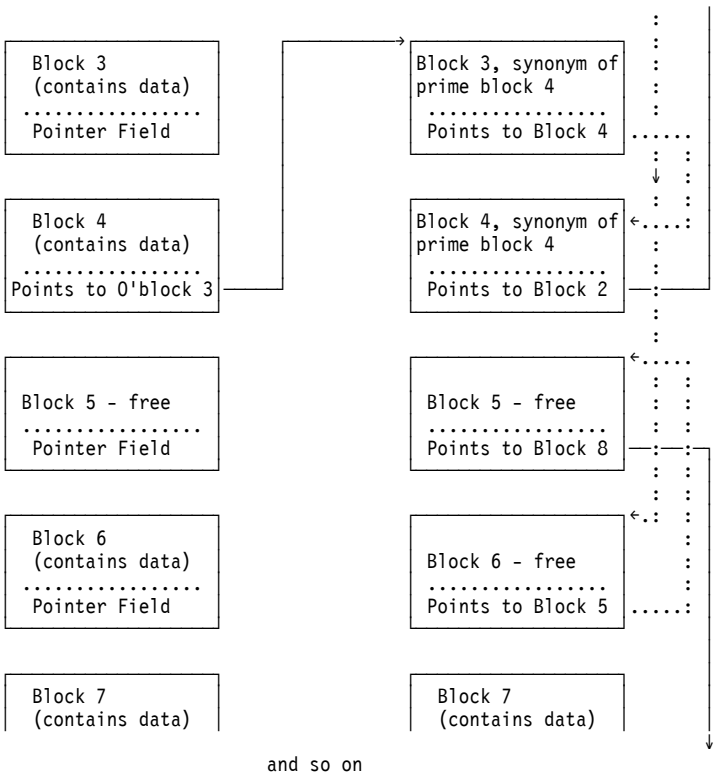


Status after Having Added a Synonym for Prime-Data Block 4



Status after Having Deleted Overflow Block 6





Summary of Methods for Randomizing

Fixed-Length Records with a Key

- Loading:
  1. Preformat the file by performing a clear disk run.
  2. Randomize to a track or a cylinder address, whichever method is used for the file. A record becomes an overflow record if the search for a dummy record is unsuccessful.
- Processing: randomize to the track or the cylinder address.

Variable-Length Records with a Key

- Loading:
  1. Do not preformat the file.
  2. Randomize to a track address. Record 0 indicates how much space is left on the track. A record becomes an overflow record if the space left on the track is too small.
- Processing: randomize to the track address.

Records without a Key

- Loading:
  1. Preformat the file by performing a clear disk run.
  2. Randomize to a record address. Each synonym becomes an overflow record to be inserted logically in an overflow chain.
- Processing:
  1. Randomize to the record address.

2. Read the record and check whether it is the one desired. If it is not, search the overflow chain.

Reading a Record

The READ macro transfers a record from Disk to the file's input area in virtual storage. You define this area to DAM by the IOAREA1 operand of the DTFDA macro for the file.

The READ macro returns control to your program after having passed the request to DAM. Your program can now perform any processing unrelated to the requested block of data. Before it performs processing related to the requested block, your program must issue a WAITF macro to check for the completion of the read operation.

The READ macro is written in either of two forms depending on the type of reference used to search for the record. In your program, you can use both forms if the records of your file has keys. The two forms are:

- If the record reference is by key

```
    READ filename,KEY
```

- If the record reference is by identifier.

```
    READ filename,ID
```

The first operand of the macro specifies the name of the file from which the record is to be read. This name is the same that you used in the DTFDA macro for the file. You can specify this operand as a symbol or in register notation. If the records of your file are undefined (RECFORM=UNDEF), DAM supplies the data length of each record in the register that you specified in the RECSIZE operand.

Record Reference by Key

This type of reference is indicated if the required control information is contained in the key area of the disk record. To retrieve records with reference by key, your DTFDA macro must include the READKEY operand.

If your program uses this method of reference, DAM requires that your program:

1. Stores the key of the desired record in the field which you specified in the DTFDA KEYARG operand.
2. Subsequently issues the READ macro.

When processing the READ macro, DAM searches the previously specified track (stored in the 8-byte track-reference field) for the desired key. If DAM finds a record with the specified key, it transfers the data area of the record to the data portion of the input area in your program.

Only the specified track is searched, except when you request that multiple tracks be searched. You do this by including the SRCHM operand in the DTFDA macro for your file. The operand causes the specified track and all following tracks to be searched until the desired record is found or the end of the cylinder is reached. The search of multiple tracks continues through the cylinder even if part of the cylinder is assigned to a different file.

If, besides the SRCHM operand, the DTFDA macro for your file includes also the IDLOC=name operand, DAM returns the ID of the record just read.

Record Reference by ID

The DTFDA macro for your file must include the READID operand.

If your program uses this method of reference, DAM requires the that your program supplies the record's track address and record number. Before it issues the READ macro, your program must store this information in the track-reference field.

When processing the READ macro, DAM searches the specified track for the desired record. When it finds a record with the specified ID, DAM transfers both the key area (if present and specified in the KEYLEN

operand of the DTFDA macro) and the data area of the record to the key and data portions of the input area in your program.

You can request DAM to return the ID of a record after a record has been read. DAM supplies this ID in the field that you defined in the IDLOC=name operand of the DTFDA macro. It returns the ID of the record that follows the record just read.

Writing a Record

A data block can be written as a new record or as an update (overwrite) of an existing record. When a new record overwrites a dummy record, DAM treats this as an update.

To overwrite an existing record, DAM requires a reference by ID or by KEY. If the records of your file are undefined (RECFORM=UNDEF is specified in the DTFDA macro), your program must do the following before it issues the WRITE macro:

1. Determine the length of each record.
2. Load this length into a register for use by DAM. This register must be defined to DAM in the RECSIZE operand of the DTFDA macro for the file.

To perform the various write operations, the WRITE macro is issued in different formats. In all cases, the WRITE macro returns control to your program after having passed the request to DAM. Your program can then perform processing unrelated to building a new block of data. Before it starts building a new block, your program must issue a WAITF macro to check for the completion of the write operation.

Adding a new Record

You accomplish this by coding a WRITE AFTER request. DAM writes the record following the one previously written on a track. (regardless of its key or ID). For DAM to do this, your program must:

1. Specify AFTER=YES in the DTFDA macro for your file.
2. Supply the track address in the field that you defined by the SEEKADR=name operand of that DTFDA macro.

If the record to be added fits on the track, DAM writes the information in the output area of your program to the track immediately following the last record on that track. For the record, DAM updates the count area and the key area (if present and specified by DTFDA KEYLEN), and the data area.

If the record does not fit or the track is not followed by enough empty tracks in the case of a spanned record, DAM does not write the record. Instead DAM sets an indication in your error/status byte specified by the DTFDA ERRBYTE operand.

If records are to be added in a disk area that contains outdated records, your program must set up record 0 of each track to reflect an empty track. You accomplish this by issuing the WRITE RZERO macro.

If you specify EOF, which applies only if you specify AFTER, DAM writes an end-of-file record (a record with a length of zero) after the last record on the track.

For a WRITE AFTER request, DAM cannot return a record ID.

If your program builds a variable-length or spanned unblocked record with a WRITE AFTER request, it must put the sum of the data length of the record to be written plus 4 into bytes 4 and 5 of the control fields preceding the data. If your program updates a record previously read from the same file by a READ macro, then do not change the control fields. Any such change causes the wrong-length-record bit to be set in the error information returned to your program.

Overwriting an Existing Record

If the reference is by ID (identifier in the count area of a record), the macro format is:

```
WRITE filename,ID
```

DAM requires, in addition, that your program:

1. Specifies WRITEID=YES in the DTFDA macro for your file.
2. Supplies both the track information and the record number in the track-reference field (which you define by the SEEKADR operand of the DTFDA macro) before the WRITE macro is issued.

DAM searches the specified track for the desired record location. When it finds the record with the specified ID, DAM transfers the information in the output area of your program to the key area (if present and specified in DTFDA KEYLEN) and the data area of the disk record. If an ID was requested, DAM returns the ID of the next record in the file.

If FIXUNB or UNDEF is specified in the RECFORM operand, then either:

- The key must precede your data in the IOAREA1 area, or
- Your program must insert the key into the key field (specified by the KEYARG operand) before you issue the WRITE macro.

This replaces the previously recorded key and data

DAM uses the count field of the original record to control the writing of the new record. A record longer than the original record is written only to the extent of the area indicated in the count field on the track; any excess bytes are lost. In that case, DAM turns on the wrong-length-record bit in the error/status field. If an updated record is shorter than the original one, DAM pads the update record with binary zeros to the length of the original record. In this case, DAM does not turn on the wrong-length-record bit.

If the reference is by key (control information is in the key area of the disk record), the macro format is:

```
WRITE filename,KEY
```

DAM requires, in addition, that your program:

1. Specifies WRITEKY=YES in the DTFDA macro for your file.
2. Supplies the key of the record to be located and the address of the track on which the record resides before the WRITE macro is issued. The key must be stored in the field that you defined in the KEYARG operand of the DTFDA macro; the track address must be stored in the track-reference field (defined in the SEEKADR operand of the DTFDA macro).

DAM searches the referenced track or, if the search-multiple-tracks option is specified in the DTFDA macro, all the remaining tracks of the same cylinder. When it finds the key, DAM writes the data without the key. This replaces the information previously recorded in the data area.

The count field of the original record controls the writing of the new record. If a record is shorter than the original one, it is padded with zeros. A record longer than the original record is written only to the extent of the area indicated in the count field on the track; any excess bytes are lost. DAM turns on the wrong-length-record bit in the error-status field if any short or long record occurs. Searching multiple tracks is specified by including the SRCHM operand in the DTFDA for your file. The search of multiple tracks continues up to the end of the cylinder even if part of the cylinder may be assigned to a different file.

If an ID is to be returned, DAM passes to your program the ID of the record following the one just written. If the search multiple-tracks option is specified, DAM returns the ID of the record just written.

Write Verification

If you specify VERIFY=YES in the DTFDA macro for your file, DAM performs a read operation after every write operation with any of the options ID, KEY, or AFTER. DAM thus checks whether the recorded data is valid.

Clearing a Track

You can erase the contents of a track by issuing a WRITE macro in the format:

```
WRITE filename,RZERO
```

Your program must supply the cylinder address and the track address. DAM expects this information in the area defined by the SEEKADR operand of the DTFDA macro for your file. DAM then:

1. Locates this track.
2. Resets the unused-bytes count (in record zero) to reflect an empty track.
3. Erases the remainder of the track.

Use this form of the WRITE macro as an initialize-disk function when your program reuses only a certain portion of a volume.

Completion of a Read or Write Operation

In your program, issue a WAITF macro to check whether a read or write operation is complete. The macro tests for exceptional conditions. It passes error/status information to the two-byte field whose name you specified in the ERRBYTE operand of the DTFDA macro for your file (the field itself must be defined in your program).

The WAITF macro requires only one operand: the name of the file for which an I/O request was issued. You can specify this operand as a symbol or in register notation.

Issue the macro before your program attempts to process an input record that has been read or starts to build another output record for your file. The macro ensures that your program does not regain control until the data transfer is complete.

Non-Data Device Operation

By a CNTRL macro with filename,SEEK specified, your program can cause access movement to begin for the next read or write operation. While the arm is moving for a SEEK, your program can process data or request I/O operations on other devices.

Note: Issuing a CNTRL macro to seek a track address may have no effect if two or more programs access the same volume, not necessarily the same file. A seek requested by one program may be made useless by another that issues an I/O request on the same volume.

DAM seeks the track that contains the next block for that file without a need to supply a track address. If the CNTRL macro is not used, DAM performs the seek operation when your program issues a READ or WRITE macro. The CNTRL macro with the SEEK operand returns control to your program as soon as the operation is initiated.

Error Handling

DAM returns to your program I/O-error codes in the two-byte field whose name you specified with the ERRBYTE operand of the DTFDA macro for your file. These codes can be tested by your program after the attempted transfer of a record is complete. That is, after DAM has returned control following the WAITF macro for the I/O request. After having examined the codes, your program can issue another I/O request macro.

One or more of the error status indication bits can be set on by DAM.

The ERREXT operand enables DAM to indicate to your program (in the ERRBYTE field) irrecoverable I/O errors (occurring before a data transfer takes place).

An explanation of the bits in the ERRBYTE field follows.

Byte 0 of ERRBYTE

Bits

Meaning / Conditions for Setting to 1

0

Reserved.

1

Wrong-length record. This bit is set on

- For fixed-length records when:

- A READ KEY or WRITE KEY is issued and the length of the key differs from the length specified by KEYLEN=n. No data is transferred.
- A READ KEY is issued and the data length differs from the specified length (BLKSIZE minus KEYLEN, or BLKSIZE minus KEYLEN plus 8 if AFTER=YES was specified).
- A READ ID is issued, and the length of the record (including key if KEYLEN was specified) differs from the specified length BLKSIZE minus KEYLEN plus 8 if AFTER=YES was specified).
- A WRITE KEY is issued, and the data length of the record is greater than specified in the count field of the record on disk. The original record position is filled; the remainder of the updated record is truncated and lost.
- A WRITE ID is issued, and the record length is greater than specified in the count field of the record on disk. The original record positions are filled; the remainder of the updated record is truncated and lost.

If an updated record is shorter than the original record, DAM pads this record with binary zeros to the original length. The wrong-length-record bit is not set on.

- For undefined records when:
  - A READ KEY or WRITE KEY is issued, and the length of the key differs from the length specified by KEYLEN=n. No data is transferred.
  - A READ KEY is issued and the data length is greater than the maximum data size (BLKSIZE minus KEYLEN, or BLKSIZE minus KEYLEN plus 8 if AFTER=YES was specified). DAM supplies the actual data length, of the record read, in the register specified by the RECSISE operand.
  - A READ ID is issued and the length of the record (including key if KEYLEN was specified) is greater than the maximum record length (BLKSIZE, or BLKSIZE minus 8 if AFTER=YES was specified). DAM supplies the actual data length, of the record read, in the register specified by the RECSISE operand.
  - A WRITE (KEY,ID, or AFTER) is issued, and the data length (loaded into the RECSIZE register) of the record is greater than the maximum data size (BLKSIZE minus KEYLEN, or BLKSIZE minus KEYLEN plus 8 if AFTER=YES was specified). The length of the record written is equal to the maximum data size.
  - A WRITE KEY is issued and the data length (loaded into the RECSIZE register) is greater than specified in the count field of the record on disk. The original record position is filled; the remainder of the updated record is truncated and lost.
  - A WRITE ID is issued, and the record length is greater than specified in the count field of the record on disk. The original record positions are filled, and the remainder of the updated record is truncated and lost.

  If an updated record is shorter than the original record, DAM pads this record with binary zeros to the original length. The wrong-length-record bit is not set on.
- For variable-length records when:
  - A READ is issued and the block length in the block descriptor is greater than the maximum value specified by the BLKSIZE operand.
  - A nonformatting WRITE is issued and the record is larger than the record on disk. The record is written with the low-order bytes truncated. The indicator is also set on if the record is shorter than the record on disk, but the low-order bytes of the record on disk are padded with binary zeros.
  - A formatting WRITE is issued and the block length in the block descriptor is greater than the maximum value specified by the BLKSIZE operand. The record is written with the low-order bytes truncated.
- For spanned records when:
  - A READ is issued and the logical record is larger than the value specified by BLKSIZE minus 8. Only the specified number of bytes has been read.

- A nonformatting WRITE is issued and the record length is not the same as that of the record being processed. If the specified length is longer than the record being processed, then the low-order bytes are ignored. If the specified length is less than the record being processed, it is padded with binary zeros.
- A formatting WRITE is issued and the size of the logical record is larger than the size specified with BLKSIZE minus 8. The record is truncated to the specified size.
- The first record on disk found by DAM is not an only or first segment. The no-record-found indicator is also set on.
- Another first segment is found by DAM after the first segment is read out and before a middle or last segment.

2

Non-data-transfer error

The block in error was neither read nor written. If ERREXT is specified and this bit is off, transfer took place. Your program should check for possible other errors in the ERRBYTE field.

3

Reserved.

4

No space available

Applies only when the WRITE AFTER form of the macro is used for a file. The bit is set on if DAM determines that there is not enough space left on the track to write the record. The record is not written.

For a spanned record the no-room-found condition is set if:

- At least one data byte does not fit on the specified track in addition to the key (if any) and the 8-byte control field, or
- Any successive tracks needed transfer the record are not completely empty.

5-6

Reserved.

7

Reference outside extents

The relative address supplied by the program is outside the extent area of the file. No I/O activity has been started, and the remaining bits should be off. If IDLOC is specified, DAM sets the field to all 9s for a zoned decimal ID and to X'FFFF' for a hexadecimal ID.

Byte 1 of ERRBYTE

Bits

Meaning / Conditions for Setting to 1

0

Data check in count area

This is an irrecoverable I/O error.

1

Track overrun

The number of bytes for the track exceeds the theoretical capacity.

2

End of cylinder

Is set on when SRCHM is specified for READ KEY or WRITE KEY and end of cylinder is reached before the desired record is found. This bit is turned on also if IDLOC=name has been specified and the

record to be read or written is the last one of the cylinder. However, the address returned by DAM is that of the first record of the next cylinder.

3

Data check when reading key or data

This is an irrecoverable I/O error.

4

No record found

A search for an ID or key is requested and DAM cannot find the requested record. Applies to both READ and and WRITE requests. May be caused by these conditions:

- The record being searched for does not exist in the file.
- The record cannot be found because of a failing seek.

For a spanned record, this bit is set on if the first physical record found by DAM is not the first or only segment.

5

End of file

Applies only if the record to be read has a data length of zero. If IDLOC is specified, DAM sets the field to all 9s for a zoned decimal ID and to X'FFFF' for a hexadecimal ID. The bit is set on after all the data records have been processed. For example, in a file of n data records, the bit is set on when your program reads the record n+1, the end-of-file record. This bit is set on also when:

- DAM finds an end-of-volume marker. Your program must determine whether this bit means end of file or end of volume.
- DAM has successfully processed a WRITE AFTER request that specified EOF.

6

End of volume

The bit is set on if the next record ID (n+1,0,1) that is returned on end of the cylinder is higher than the volume address limit. If both the end-of-cylinder and end-of-volume indicators are set on and IDLOC is specified, DAM sets the IDLOC field to all 9s for a zoned decimal ID and to X'FFFF' for a hexadecimal ID.

7

Reserved.

Closing the File

This section concludes the discussion of programming with the macros available for direct processing of the records of a file.

In your program, use the CLOSE macro after the processing of the file is complete. The macro ends the association of the logical file defined in your program with your file of data on disk.

The CLOSE macro deactivates a file that was previously opened with an OPEN macro for the file. If trailer labels are specified, DAM writes them on output and makes them available for checking by your program on input.

A file may be closed at any time by issuing this macro. Once your program has issued a CLOSE macro for a file, it cannot issue any further I/O request for the file until it is reopened.

Use CLOSER if the file was activated by an OPENR macro.

Appendix C. Processing a File with Physical IOCS (PIOCS)

Your program can access a file by using PIOCS macros such as EXCP and CCB. However, a file must first be defined by the DTFPH macro, if this is a file on disk or on tape with standard labels. You must define a file on disk by way of the DTFPH macro. For example, if checkpoint records are to be written into this file. The information in this appendix is limited to the access of such files.

A logic module is not required. However, your program must include a channel program for the file that is to be accessed. You define the address of this channel program in the CCB macro for the file.

In PIOCS, the logical unit name is specified in the CCB or IORB and also in the DTFPH macro. In addition, it is specified in the EXTENT job control statement. If more than one of these specifications is used, then:

- The specification in the EXTENT statement overrides the specification in the DTFPH macro.
- The specification in the CCB or IORB macro overrides the specification in an EXTENT statement and in the DTFPH macro.

Figure 71 on page 211 shows the relationship between the source program and the job control I/O assignment.

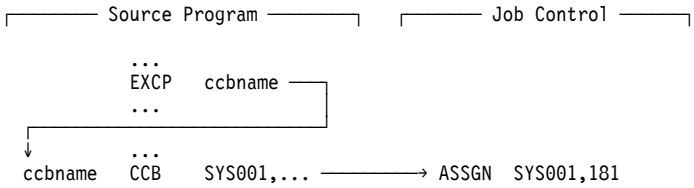


Figure 71. Relationship Between Source Program and Job Control I/O Assignment

After a file has been defined by a DTFPH macro, the imperative macros can be used to operate on the file. Imperative macros are available for opening, processing, and closing a file.

Opening the File

The OPEN macro activates a file that is defined by a DTFPH macro.

The macro associates the logical file as defined in your program with a certain file of data on a disk volume, for example. The file remains active until your program issues a CLOSE macro for the file.

If OPEN attempts to activate a file whose device is unassigned, IOCS cancels the job. If the device is assigned IGN, OPEN does not activate the file. Instead, OPEN sets on bit 2 of byte 16 of the DTF table to indicate that the file is inactive. Do not attempt any I/O operations for an inactive file.

In your program, you can open up to 16 files with one OPEN macro. These files may use any combination of access methods. You specify the name of the file to be opened either symbolically or by using register notation.

If, for a file on disk or tape, you plan to process user-standard labels (UHLn or UTLn) or nonstandard tape labels, you must provide the information for checking or building the labels. If this information is to be retrieved from another input file, your program must open that file before it opens the disk or tape file. Do this by either:

- Specifying the name of the input file ahead of the tape or disk file in the same OPEN, or
- Issuing a separate OPEN for the input file preceding the open for your tape or disk file.

If an output file on tape is specified to have standard labels and OPEN finds no volume label, PIOCS writes a message to the console. Your operator can then supply a volume serial number for PIOCS to write a volume label onto the tape.

The actions of PIOCS in response to an OPEN macro vary slightly for the types of files as described below.

Disk Volumes – Output

Single Volume Mounted

For the volume, OPEN checks the VOL1 label and the extents specified in the EXTENT job control statements:

- The extents must not overlap each other.
- If user-standard header labels are written, the first extent must be at least two tracks long.
- Extents of type 1 and 8 are valid for a CKD disk; only extents of type 1 are valid for an FBA disk.

OPEN checks all the labels in the VTOC to ensure that the file to be created does not destroy a file whose expiration date is still pending. After this check, OPEN creates the standard label(s) for the file and writes the label(s) into the VTOC.

To create your own user-standard header labels (UHLn) for the file, you must include the LABADDR operand in the DTFPH macro. OPEN reserves the first track of the first extent for these labels and gives control to your label routine. When the processing of user-standard labels is complete, the first extent of the file can be used.

When all processing for an extent is complete, your program must issue another OPEN for the file to make the next extent available. When the last extent on the last volume of the file has been processed, the next OPEN does one of the following:

- If an end-of-extent exit routine was defined to PIOCS (by the DTFPH operand EOXPTR=name), OPEN passes control to this routine. In addition, OPEN stores the character F at filename+30. Your exit routine can check this byte.
- If an end-of-extent routine is not defined, OPEN issues a message to the console. Your system operator now can either:
  - Cancel the job, or
  - Supply an extent and have the system continue processing the job.

If the system provides DASD file protection, only the extents opened for the mounted volume are available to your program.

All Volumes Mounted

Each volume of your file is opened before the file is processed.

For each volume, OPEN checks the standard VOL1 label and checks the extents specified in the EXTENT job control statements for the following:

1. The extents must not overlap each other.
2. Only extents of type-1 can be used.
3. If user-standard header labels are created, the first extent must be at least two tracks long.
4. All volumes of a file must be of the same type and model.

OPEN checks all the labels in the VTOC to ensure that the file to be created will not write over a file that has not yet expired. After this check, OPEN creates the standard label(s) for the file and writes the label(s) in the VTOC.

If you wish to create your own user-standard header labels for the file, include the LABADDR operand in the DTF. OPEN reserves the first track of the first extent for these labels and gives control to your label routine. When your program has built the last label, return control to OPEN by issuing a LBRET 1 macro. OPEN then continues to open the next volume. After all volumes are opened, the file is ready for processing.

If you specified the XTNTXIT operand in the DTFPH macro for the file, OPEN gives control to your extent-exit routine. OPEN passes to this routine the address of a 14-byte extent information area in register 1. Your program can save the contents of this area and use the information later on for specifying record addresses. If your disk file is file protected, your program cannot write into any extents while your XTNTXIT routine has control.

Disk Volumes – Input

Single Volume Mounted

OPEN is required only if it is desirable to have the standard labels for the file to be checked.

All Volumes Mounted

When all volumes containing the input file are online and ready at the same time, each volume is opened, one at a time, before any processing is done. OPEN checks for each volume:

- The extents specified in the EXTENT job control statements
- The standard VOL1 label.
- The file label(s) in the VTOC.

If LABADDR is specified, OPEN makes the user-standard header labels (UHL) available to your program for checking, one after the other, one at a time. When checking is complete, return control to OPEN by issuing the LBRET 2 macro, which opens the next volume. After all volumes are opened, the file is ready for processing.

If you specified the XTNTXIT operand in your DTFPH macro, OPEN gives control to your extent-exit routine. OPEN passes to this routine the address of a 14-byte extent information area in register 1. Your program can save the contents of this area and use the information later on for specifying record addresses.

Processing of User Labels and Extent Information

This section applies to disk and tape devices for the processing of user labels; it applies to disk devices for the processing of extent information.

To do this processing, your program must include a routine whose address you define in your DTFPH macro by the operand

LABADDR=name
For the processing of labels.

XTNTXIT=name
For the processing of extent information.

Checking of User-Standard Labels on Disk

IOCS checks only header labels; it does not check trailer labels.

It passes labels to your program, one after the other and one at a time, until the maximum allowable number is read and updated, or until your program indicates that it wants no more. In your label routine, use:

- LBRET 3 if you want IOCS to update (rewrite) the label just read and pass the next label.
- LBRET 2 if you want IOCS to read and pass the next label.
- LBRET 1 if none of the remaining labels (if any) are to be checked.

If IOCS finds an end-of-file record while your program uses LBRET 2 or LBRET 3, IOCS ends the checking of labels.

Writing of User-Standard Labels on Disk

In your routine, build the labels, one at a time and one after the other. Each time you have finished building a label, return control to IOCS to write the label. To return control, use:

- LBRET 2 if you wish to regain control after IOCS wrote the label.
- LBRET 1 to stop writing labels before the maximum number of labels is written.

If IOCS finds that the maximum number of labels has been written, IOCS ends further processing of labels.

Checking of Extents

For an input file with all volumes mounted, your program can process your extent information. After an extent is processed, use in your XTNTXIT routine:

- LBRET 2 to have IOCS pass to your routine the information for the next extent.
- LBRET 1 to return control to IOCS when extent processing is complete.

Checking of User-Standard Labels on Tape

Physical IOCS reads and passes the labels to your LABADDR routine, one after the other and one at a time. IOCS stops passing labels when either:

- It finds a tapemark, or
- You indicate to IOCS that you want no more labels. To do this, return control to IOCS by using LBRET 1.

Use LBRET 2 if you want to process the next label.

Writing of User-Standard Labels on Tape

In your LABADDR routine, build the labels, one after the other and one at a time. Each time you have completed building a label, return control to IOCS by a LBRET 2 macro. This causes IOCS to write the label.

Note: Your program must accumulate the correct block count, if this is desired. Move this count (for inclusion in the standard trailer label) to the four-byte field named `filenameB`. Provide this count in binary form.

Use LBRET 1 to have PIOCS end the processing of labels.

Writing or Checking of Nonstandard Tape Labels

In your LABADDR routine, you must process all your nonstandard labels at once. Use LBRET 2 after all label processing is completed and you want to return control to IOCS.

Reading and Writing of Records

An I/O request to PIOCS requires, in your program:

- A channel program of one or more CCWs
- An I/O control block (CCB or IORB) that points to the first (or only) CCW of your channel program
- An execute-channel-program (EXCP) macro referring to the I/O block.

Your program must wait for completion of the requested I/O before it can process the data read (on input) or prepare new data for being written (on output).

The Command Control (I/O-Request) Block

For the I/O device to be accessed, PIOCS requires that your program includes a command control block (CCB) or an I/O request block (IORB). PIOCS uses the block to communicate with your program about such things as the status of the device and the operations to be performed.

The CCB macro generates a CCB of 16 or 24 bytes. Similarly, the IORB and GENIORB macros generate an IORB. An IORB is the same as a CCB, except that bytes 6 through 12 and 16 through 23 are reserved for use by IOCS. Using the IORB or GENIORB macros instead of the CCB macro allows you to specify additional options such as areas to be page-fixed. This frees the system from having to find out which areas are to be fixed.

A CCB (or IORB) macro cause the control block to be assembled into your program. The GENIORB macro causes the control block to be set up while your program runs. See z/VSE System Macros Reference for the format of the CCB, IORB, and GENIORB macros and for the layout and contents of a CCB or an IORB macro. The publication describes in detail the transmission-information and user-option bits in bytes 2 and 3 of these blocks.

Only some of the user-option bits can be set on when you use the IORB or GENIORB macros; you do this by specifying the IOFLAG operand. By way of the CCB macro you can set any or all of the option bits. If more than one option bit must be set, use the sum of the values. For example, to set on user option bits 3, 5, and 6 of byte 2, use X'1600':

```
X'1000' + X'0400' + X'0200' = X'1600'
```

For certain I/O devices, certain user option bits may have to be set on. Assume a CCB for a disk or a tape device, and your program is to receive control on a read data check. In this case, option bit 6 of byte 2 (return) must be set on.

If command chaining is used in the channel programs for these devices, option bit 7 of CCB byte 3 (command chain retry) must also be set on.

If your program has its own error routine (bit 7 of CCB byte 2 is on, or X'nnnn' in the CCB macro is X'0100') but has not specified a sense address in the CCB macro, the system clears the sense information to prevent possible deadlocks in the control unit. If then your error processing routine issues an EXCP to do a sense operation, the information about the unit check has already been cleared.

The Execute Channel Program (EXCP) Macro

The macro initiates an I/O request. The assembler translates the macro into an SVC instruction with a reference to the CCB or IORB whose address is specified in the macro. In the macro, this address can be given as a symbol or in register notation.

PIOCS determines the actual device from the CCB (or IORB). It causes the channel program to be processed as soon as the channel and the device are available. Interruptions for I/O completion are used to control Start I/O requests, if the channel or device was busy when PIOCS processed the EXCP macro.

The WAIT Macro

PIOCS does not wait for the completion of an I/O operation. Instead, it returns control to your program after having started the operation. Your program must ensure that it does not start processing a block of data that has not been completely read, or start overwriting an output area before the previous block has been completely written. To accomplish this, code a WAIT macro in your program. The macro must refer to the applicable CCB or IORB. This reference can be by the symbolic name of the control block or by register notation. The WAIT macro causes the system to check the status of the pending I/O operation.

When your program issues a WAIT, the supervisor gives control to another program until the traffic bit (bit 0 of byte 2) of the related CCB or IORB is turned on.

Figure 72 on page 216 shows the relationship between the macros CCB, EXCP, and WAIT. It shows also the assembler instruction (not macro) CCW, which is included to give you a complete picture. An IORB or a GENIORB macro can be used instead of the CCB in some cases.

```
                ...
               EXCP    MYCCB
                ...
               WAIT    MYCCB
                ...
                ...       Other (non-I/O related) code
                ...
MYCCB          CCB     SYS015,MYCHPRG,...
                ...
MYCHPRG        CCW     cc,data-addr,flags,count
               CCW     cc,data-addr,flags,count
               CCW     cc,data-addr,flags,count
               ...
```

Figure 72. Relationship Between the PIOCS Macros (No DTFPH Used)

Additional Macros

EXTRACT ID=PUB
The macro retrieves partition-related device information. By using the macro, your program can determine the type of the device to which a logical unit is assigned.

For a given logical unit the macro returns physical-unit-block (PUB) information. To interpret this information, use the mapping DSECT generated by the IJBPUB macro in your program.

SECTVAL
The macro calculates the sector value of the address of the requested record on the track of a disk device. The macro returns this value in register 0.

The system calculates this value from the information supplied by your program: data length, key length, record number, and the device type.

You need the macro if your program makes use of rotational position sensing (RPS).

Forcing an End-of-Volume Condition

It might be desirable to stop the processing of records for a tape file before IOCS detects that the physical end of the volume is near. To do this, issue:

- The FEOV (force end of volume) macro if the associated tape drive is assigned to a programmer logical unit. The macro indicates to IOCS that:
  1. No more records of your tape file are to be read from or written to the currently used volume.
  2. More records of the same logical file are to be read from or written to another volume.

  If also standard labels are to be processed, FEOV can be issued for output data files only. In this case, FEOV:
  1. Writes a tapemark.
  2. Writes the standard trailer label.
  3. If a label-exit routine is specified (by the LABADDR=name operand), writes any user-standard trailer labels.

  When the new volume is mounted and ready for output, PIOCS writes the standard header label and user-standard header labels, if any.
- The SEOV (system end-of-volume) macro if the associated tape drive is assigned to SYSLST or SYSPCH. The macro:
  1. Writes a tapemark.
  2. Rewinds and unloads the tape.
  3. Checks for an alternate tape. If it cannot find an alternate tape, IOCS writes a message to the console. The operator can now mount a new tape on the same drive and continue. If it finds an alternate tape, IOCS opens the new tape and makes it ready for processing.

To use this macro, your program must check for the end-of-volume condition in the associated CCB.

Closing the File

The CLOSE macro is used to deactivate any file that was previously opened. Console files, however, cannot be closed.

The macro ends the association of the logical file defined in your program with a file of data on an I/O device.

No further I/O requests should be issued for a closed file until it is opened again.

Up to 16 files can be closed by one macro by supplying additional file names as operands. Instead of coding the name of the file that is to be closed, you can use ordinary register notation. Your program must, before it issues the macro, load the address of the stored name of the file into the specified register.

Hints for Programming

The following sections discuss some of the problems that you might encounter when you use physical IOCS to code some of your program's input and output. Some restrictions are also mentioned.

LIOCS Functions for Processing with PIOCS

In general, your program must provide all of the logical functions that are normally provided by LIOCS. However, your program must make use of LIOCS if:

- The file resides on disk.
- The file resides on tape and requires the processing of labels.

For such files, LIOCS performs label processing as usual in response to the OPEN and CLOSE macros. In addition, your program can issue the FEOV macro for volume switching on magnetic tape output files.

If the DTFPH macro is used, a program might look slightly different from the example that is given in Figure 72 on page 216. As mentioned earlier, the DTF table contains the CCB in the first 16 bytes, so that the EXCP and WAIT macros can now refer to the name of the DTFPH macro. The DTFPH macro in turn must include the operand CCWADDR=name so that the CCB has the proper reference to the first CCW of the channel program that is to be used. This is shown in Figure 73 on page 217.

```
PIOCSFL   DTFPH  CCWADDR=FRSTCCW,DEVADDR=SYS011,...
           ...
          OPEN    PIOCSFL
           ...
          EXCP    PIOCSFL
           ...
          WAIT    PIOCSFL
           ...
          CLOSE   PIOCSFL
           ...
FRSTCCW   CCW     cc,data-addr,flags,count
          CCW     cc,data-addr,flags,count
           ...
```

Figure 73. Relationship Between DTFPH and Other PIOCS Macros

Command-Chain Retry

For information about the CCW format and the concepts of data and command chaining, refer to your *Principles of Operations* publication.

You can use the command-chain-retry option for your channel programs by setting on the command-chain-retry bit in the CCB. If an error involving a retry occurs, the retry begins with the CCW executed last. If the bit is off, the entire channel program is retried.

To make use the command-chain-retry option, your program must move the address of the first CCW of the channel program to bytes 9 - 11 of the CCB before it issues an EXCP. This ensures that the CCB always

contains the correct CCW address. Bytes 9 - 11 are modified by PIOCS for a retry after an error with the address of the CCW to be retried. These bytes are not reset to their original contents.

A command chain might be broken by an exceptional condition that does not cause PIOCS to restart the channel program (wrong-length record or unit exception, for example). In this case, you can determine the address of the CCW executed last and, if necessary, restart the channel program at that point. To obtain this address, subtract 8 from the address in bytes 13 - 15 of the CCB.

On a 1403 printer, a command chain is broken after sensing channel 9 or 12. When using command chaining on such a printer, your program should therefore always check whether the entire CCW chain has been executed.

Do not use the command-chain-retry bit for:

- Reading multiple blocks from SYSIPT or SYSRDR.
- Channel programs to read from or write to a disk.

Channel Indirect Data Addressing

This is specified by a bit in the CCW. When on, the bit indicates that the data address in the CCW points to a list of words called indirect-data-address words (IDAWs). Each of these words contains an absolute address designating a data area in storage.

When the indirect-data-addressing bit in the CCW is on, bits 8 - 31 of the CCW point to the first IDAW to be used for data transfer. Additional IDAWs, if needed for completing the data transfer for the CCW, follow the first IDAW, one after the other. The number of IDAWs needed for a CCW is determined by the count field of the CCW and by the data address in the first IDAW.

Make use of channel-indirect-data addressing only if you use also EXCP REAL.

Data Chaining

If you use data chaining, all of the CCWs in your channel program should include the command code for the operation that is to be performed. This ensures proper I/O error recovery.

When no error occurs, IOCS ignores the command code if a preceding CCW has the data chaining bit on. In case of an error, however, recovery frequently depends on the command being executed, and the command code in the last CCW is examined. In such a case, a 'dummy' command code prevents error recovery.

Channel Program for a File on a CKD Disk

Always start your channel program with a full seek (command code X'07'). If the channel program includes embedded seeks, they should be full seeks as well.

A program using embedded full seeks cannot run with DASD file protection, nor can it take full advantage of the seek separation feature. With DASD file protection, an embedded full seek causes the program to be canceled.

The seek separation feature initiates a seek and separates it from the channel program chain. Thus, the channel is available for other input or output operations. However, the feature works only for the first seek of a channel command chain. When executing a channel program, the system's supervisor sets up a channel program with three commands (see also Figure 74 on page 219):

1. A Seek that is identical to the user's seek.
2. A Set File Mask that prevents other X'07' seeks from being executed.
3. A Transfer in Channel (TIC) command that transfers control to the command following the user's seek.

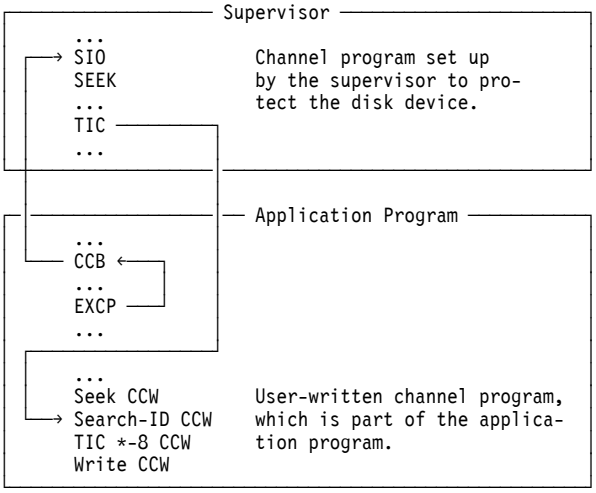


Figure 74. Example of Channel Programming a File-Protected CDK DASD File

RPS (Rotational Position Sensing)

If a system has been generated to support RPS, the user can include the channel commands set sector and read sector for DASDs supporting the feature. These commands can be used (1) to determine the distance between the requested record and the read/write head and (2) to free the channel for other operations until the requested record is under the read/write head.

The SECTVAL macro can be used to calculate a sector value for a specified record.

Channel Program for an FBA Device

Access to data on an FBA disk is accomplished by a channel program in three basic steps:

1. The execution of a DEFINE EXTENT command (command code X'63'). There is no SEEK CCW for FBA devices.

   The DEFINE EXTENT command defines the location and size of a data extent; thus it establishes the bounds on disk within which subsequent chained commands are permitted to operate. The command includes a file protect mask for controlling the execution of following commands.
2. The execution of a Locate command (command code X'43'). Iit specifies the location and number of addressable blocks of the data space to be processed and the operation (read or write) to be performed.
3. When the storage device is positioned for a data transfer, a READ or WRITE command causes the transfer of data between the storage device and main storage. A Read or Write command must be chained from a Locate command. If the chaining prerequisite is not satisfied, the command is rejected with a unit check (command reject). Although Read (Write) commands may not be command chained from another Read (Write), data chaining of Read (Write) CCWs is permitted. However, data chaining within a block may cause overruns or chaining checks.

Console (Printer-Keyboard) Buffering

If a printer-keyboard is assigned to SYSLOG, you can increase throughput on output for records that do not exceed 80 characters. Start the I/O command and return to your program before the output is completed.

Blocks are always printed in a FIFO (first-in-first-out) order, regardless of whether or not the output blocks are buffered (queued on an I/O completion basis).

Console buffering is performed on output only if:

- The block to be written is not longer than 80 characters.
- No data chaining or command chaining is performed.

- The CCB is not set to:
  - Accept irrecoverable I/O errors
  - Post at device end
  - Pass control to a user error routine
  - Provide sense information.

Alternate Tape Switching

Alternate tape drives cannot be used on input from tape processed by PIOCS.

On output, automatic alternate tape drive switching can be done through the DTFPH and FEOV macros. The FEOV macro writes the trailer labels and deactivates the currently accessed volume. IOCS expects the next (new) volume to be mounted on the alternate drive. IOCS writes header labels as required on the new volume.

Bypassing of Embedded Checkpoints on Tape

Checkpoint information is written as a set of records as follows:

- One 20-byte header record
- One status descriptor record containing system-status information.
- As many records as are needed to save the contents of the selected parts of virtual storage.
- One 20-byte trailer record.

Depending on whether the file is processed forward or backward, the header or trailer record can be used to recognize and bypass checkpoint sets. The format of both the header and the trailer record is:

Bytes
Contents

00-11
///bCHKPTb// – Note the two blanks represented by the letter b, one before and one after the string CHKPT.

12-13
The number (in binary) of checkpoint records containing program data.

14-15
The total number (in binary) of checkpoint records.

16-19
The serial number of the checkpoint taken.

Checkpoint information can always be identified by the first 12 bytes of the header or trailer record (depending on whether the file is read forward or backward).

If the file is read forward, the checkpoint header record occupies the 20 high-order bytes of the I/O area. If the file is read backward, the checkpoint trailer record occupies the 20 low-order bytes of the I/O area.

To bypass checkpoint information:

1. Go into a read loop, until a checkpoint trailer record (reading forward) or header record (reading backward) is encountered.
2. Extract bytes 14 and 15 from the header or trailer record and space forward or backward that number of records.

You can use read commands or forward-space commands, whichever is more convenient.

Restrictions for the IBM 3800 Printing Subsystem

Before using PIOCS on this device, consult the *IBM 3800 Programmer's Guide* about restrictions that should be observed when you code your channel programs. This prevents errors during subsequent jobs.

Appendix D. Using System Control Macros in Reenterable Programs

A reenterable program, if properly coded, can be used concurrently by several tasks without sacrificing the integrity of its instructions or data areas.

Data areas that may be modified by a reenterable program must be unique to each task using that program. Examples of such areas are save areas, I/O areas, and control blocks. Consider the program shown below:

```
                                          Column 72
          ATTACH  SUBTASK,SAVE=LOCSAV,         X
                  ECB=LOCECB,                  X
                  ABSAVE=LOCSAVAB
           ...
          WAIT    LOCECB
           ...
LOCSAV    DS      16D
LOCSAVAB  DS      CL(SVUOLDLN)
LOCECB    DC      F'0'
           ...
          MAPSAVAR  Mapping of exit save area
```

A task that executes the above ATTACH macro initializes the save area (LOCSAV) for the subtask to be attached. After that subtask has started processing, it can be interrupted. While the subtask remains in the wait state, another task may be dispatched and execute the macro. Because only one subtask save area exists, this other task, in initializing the save area, would destroy whatever was saved there when the interrupt occurred.

As coded in the example, the ATTACH macro is not reenterable. The involved data areas are not unique to the tasks running the macro.

A commonly used method of isolating data areas for individual tasks is to set up these areas outside the program's boundaries. Through the GETVIS macro, a task can dynamically acquire storage, which it can use as a data area. This data area can be kept unique to the task that uses it.

Dynamically acquired storage areas are addressable via registers. Figure 75 on page 222 shows how to dynamically acquire and address storage and how to refer to the individual fields of that storage in the ATTACH macro.

```
                                                                  Column 72
T06BMAIN  CSECT
           ...
DYNSTOR   DSECT    ,            DYNAMIC STORAGE AREA
DYNPARM   DS       CL64         REENTERABLE MACRO PARM AREA
DYNSAV    DS       CL128        SUBTASK SAVE AREA
DYNSAVAB  DS       CL(SVUOLDLN)   SUBTASK AB-EXIT SAVE AREA
DYNECB    DS       F            SUBTASK ECB
DYNSTORL  EQU      *-DYNSTOR    LENGTH OF DYN STORAGE
          MAPSAVAR               MAPPING OF EXIT SAVE AREA
T06BMAIN  CSECT
           ...
          LA       0,DYNSTORL   LENGTH FOR GETVIS
          GETVIS   ADDRESS=(10)
           ...
          USING    DYNSTOR,10   MAKE DYNAMIC STORAGE
*                               ADDRESSABLE THRU BASE REG.10
          LA       7,DYNSAV
          LA       8,DYNECB
          LA       9,DYNSAVAB
          ATTACH   SUBTASK,SAVE=(7),ECB=(8),ABSAVE=(9),           X
                   MFG=(10)
           ...
```

Note: MFG stands for "Macro Format Generate."

Figure 75. Dynamically Acquiring Storage and Addressing this Storage

The MFG operand in the macro causes VSE to dynamically build the parameter list outside the macro expansion. The MFG operand points to this list, which is a 64-byte area that is provided by the program for the execution of the macro.

For macros that allow the MFG operand to be specified, refer to z/VSE System Macros Reference.

Register notation as used in the preceding example can be costly and cumbersome to code. Each operand uses a register and, in addition, each register has to be preloaded with the address of the corresponding field.

Where indicated in the macro format (described in z/VSE System Macros Reference) , the operand can be specified in the (S,address) notation rather than in register notation. The ATTACH macro is one of the macros that allows you to do this:

```
                                              Column 72

  ATTACH     SUBTASK,                             X
             SAVE=(S,DYNSAV),                     X
             ECB=(S,DYNECB),                      X
             ABSAVE=(S,DYNSAVAB),                 X
             MFG=(S,DYNPARM)
```

An operand that is written in (S,address) notation assembles like an S-type address constant: its object code is an assembler instruction address in base register/displacement form. For example, for the above ATTACH macro (assuming the example in Figure 75 on page 222), the reference:

- DYNSAV assembles into X'A040',
- DYNECB assembles into X'A0D0'.

Both addresses use the same base register: register 10.

With (S,address) notation, only one register is used, the one that serves as the base register for the DSECT.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/VSE.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IPv6/VSE is a registered trademark of Barnard Software, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein. IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed. You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/VSE enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using Assistive Technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/VSE. Consult the assistive technology documentation for specific information when using such products to access z/VSE interfaces.

Documentation Format

The publications for this product are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF files and want to request a web-based format for a publication, you can either write an email to s390id@de.ibm.com, or use the Reader Comment Form in the back of this publication or direct your mail to the following address:

```
IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Glossary

This glossary includes terms and definitions for IBM z/VSE.

The following cross-references are used in this glossary:

1. See refers the reader from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
2. See also refers the reader to a related or contrasting term.

A

Access Control Logging and Reporting

An IBM licensed program to log all attempts of access to protected data and to print selected formatted reports on such attempts.

access control table (DTSECTAB)

A table that is used by the system to verify a user's right to access a certain resource.

access list

A table in which each entry specifies an address space or data space that a program can reference.

access method

A program, that is, a set of commands (macros) to define files or addresses and to move data to and from them; for example VSE/VSAM or VTAM.

account file

A disk file that is maintained by VSE/POWER containing accounting information that is generated by VSE/POWER and the programs running under VSE/POWER.

addressing mode (AMODE)

A program attribute that refers to the address length that a program is prepared to handle on entry. Addresses can be either 24 bits, 31 bits, or 64 bits in length. In 24 bit addressing mode, the processor treats all virtual addresses as 24-bit values; in 31 bit addressing mode, the processor treats all virtual addresses as 31-bit values and in 64-bit addressing mode, the processor treats all virtual addresses as 64-bit values. Programs with an addressing mode of ANY can receive control in either 24 bit or 31 bit addressing mode. 64 bit addressing mode cannot be used as program attribute.

administration console

In z/VSE, one or more consoles that receive all system messages, except for those that are directed to one particular console. Contrast this with the user console, which receives only those messages that are directed to it, for example messages that are issued from a job that was submitted with the request to echo its messages to that console. The operator of an administration console can reply to all outstanding messages and enter all system commands.

alternate block

On an FBA disk, a block that is designated to contain data in place of a defective block.

alternate index

In systems with VSE/VSAM, the index entries of a given base cluster that is organized by an alternate key, that is, a key other than the prime key of the base cluster. For example, a personnel file preliminary ordered by names can be indexed also by department number.

alternate library

An interactively accessible library that can be accessed from a terminal when the user of that terminal issues a connect or switch library request.

alternate track

A library, which becomes accessible from a terminal when the user of that terminal issues a connect or switch (library) request.

AMODE

Addressing mode.

APA

All points addressable.

APAR

Authorized Program Analysis Report.

appendage routine

A piece of code that is physically located in a program or subsystem, but logically and extension of a supervisor routine.

application profile

A control block in which the system stores the characteristics of one or more application programs.

application program

A program that is written for or by a user that applies directly to the user's work, such as a program that does inventory control or payroll. See also batch program and online application program.

AR/GPR

Access register and general-purpose register pair.

ASC mode

Address space control mode.

ASI (automated system initialization) procedure

A set of control statements, which specifies values for an automatic system initialization.

attention routine (AR)

A routine of the system that receives control when the operator presses the Attention key. The routine sets up the console for the input of a command, reads the command, and initiates the system service that is requested by the command.

automated system initialization (ASI)

A function that allows control information for system startup to be cataloged for automatic retrieval during system startup.

autostart

A facility that starts VSE/POWER with little or no operator involvement.

auxiliary storage

Addressable storage that is not part of the processor, for example storage on a disk unit. Synonymous with external storage.

B

B-transient

A phase with a name beginning with $$B and running in the Logical Transient Area (LTA). Such a phase is activated by special supervisor calls.

bar

2 GigyByte (GB) line

basic telecommunications access method (BTAM)

An access method that permits read and write communication with remote devices. BTAM is not supported on z/VSE.

BIG-DASD

A subtype of Large DASD that has a capacity of more than 64 K tracks and uses up to 10017 cylinders of the disk.

block

Usually, a block consists of several records of a file that are transmitted as a unit. But if records are very large, a block can also be part of a record only. On an FBA disk, a block is a string of 512 bytes of data. See also a control block.

block group

In VSE/POWER, the basic organizational unit for fixed-block architecture (FBA) devices. Each block group consists of a number of 'units of transfer' or blocks.

C

CA splitting

Is the host part of the VSE JavaBeans, and is started using the job STARTVCS, which is placed in the reader queue during installation of z/VSE. Runs by default in dynamic class R. In VSE/VSAM, to double a control area dynamically and distribute its CIs evenly when the specified minimum of free space get used up by more data.

carriage control character

The fist character of an output record (line) that is to be printed; it determines how many lines should be skipped before the next line is printed.

catalog

A directory of files and libraries, with reference to their locations. A catalog may contain other information such as the types of devices in which the files are stored, passwords, blocking factors. To store a library member such as a phase, module, or book in a sublibrary. See also VSE/VSAM catalog.

cell pool

An area of virtual storage that is obtained by an application program and managed by the callable cell pool services. A cell pool is located in an address space or a data space and contains an anchor, at least one extent, and any number of cells of the same size.

central location

The place at which a computer system's control device, normally the systems console in the computer room, is installed.

chained sublibraries

A facility that allows sublibraries to be chained by specifying the sequence in which they must be searched for a certain library member.

chaining

A logical connection of sublibraries to be searched by the system for members of the same type (phases or object modules, for example).

channel command word (CCW)

A doubleword at the location in main storage that is specified by the channel address word. One or more CCWs make up the channel program that directs data channel operations.

channel program

One or more channel command words that control a sequence of data channel operations. Execution of this sequence is initiated by a start subchannel instruction.

channel scheduler

The part of the supervisor that controls all input/output operations.

channel subsystem

A feature of z/Architecture that provides extensive additional channel (I/O) capabilities to IBM Z.

channel to channel attachment (CTCA)

A function that allows data to be exchanged

1. Under the control of VSE/POWER between two virtual VSE machines running under VM or
2. Under the control of VTAM between two processors.

character-coded request

A request that is encoded and transmitted as a character string. Contrast with *field-formatted request*.

checkpoint

1. A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.
2. To record such information.

CICS (Customer Information Control System)

An IBM program that controls online communication between terminal users and a database. Transactions that are entered at remote terminals are processed concurrently by user-written application programs. The program includes facilities for building, using, and servicing databases.

CICS ECI

The CICS External Call Interface (ECI) is one possible requester type of the *CICS business logic interface* that is provided by the CICS Transaction Server for z/VSE. It is part of the CICS client and allows workstation programs to CICS function on the z/VSE host.

CICS EXCI

The EXternal CICS Interface (EXCI) is one possible requester type of the *CICS business logic interface* that is provided by the CICS Transaction Server for z/VSE. It allows any BSE batch application to call CICS functions.

CICS system definition data set (CSD)

A VSAM KSDS cluster that contains a resource definition record for every record defined to CICS using resource definition online (RDO).

CICS Transaction Server for z/VSE

A z/VSE base program that controls online communication between terminal users and a database. This is the successor system to CICS/VSE.

CICS TS

CICS Transaction Server

CICS/VSE

Customer Information Control System/VSE. No longer shipped on the Extended Base Tape and no longer supported, cannot run on z/VSE 5.1 or later.

class

In VSE/POWER, a group of jobs that either come from the same input device or go to the same output device.

CMS

Conversational monitor system running on z/VM.

common library

A library that can be interactively accessed by any user of the (sub)system that owns the library.

communication adapter

A circuit card with associated software that enables a processor, controller, or other device to be connected to a network.

communication region

An area of the supervisor that is set aside for transfer of information within and between programs.

component

1. Hardware or software that is part of a computer system.
2. A functional part of a product, which is identified by a component identifier.
3. In z/VSE, a component program such as VSE/POWER or VTAM.
4. In VSE/VSAM, a named, cataloged group of stored records, such as the data component or index component of a key-sequenced file or alternate index.

component identifier

A 12-byte alphanumeric string, uniquely defining a component to MSHP.

conditional job control

The capability of the job control program to process or to skip one or more statements that are based on a condition that is tested by the program.

connect

To authorize library access on the lowest level. A modifier such as "read" or "write" is required for the specified use of a sublibrary.

connection pooling

Introduced with an z/VSE 5.1 update to manage (reuse) connections of the z/VSE database connector in CICS TS.

connector

In the context of z/VSE, a connector provides the middleware to connect two platforms: Web Client and z/VSE host, middle-tier and z/VSE host, or Web Client and middle-tier.

connector (e-business connector)

A piece of software that is provided to connect to heterogeneous environments. Most connectors communicate to non-z/VSE Java-capable platforms.

container

Is part of the JVM of application servers such as the IBM WebSphere Application Server, and facilitates the implementation of servlets, EJBs, and JSPs, by providing resource and transaction management resources. For example, an EJB developer must not code against the JVM of the application server, but instead against the interface that is provided by the container. The main role of a container is to act as an intermediary between EJBs and clients, Is the host part of the VSE JavaBeans, and is started using the job STARTVCS, which is placed in the reader queue during the installation of z/VSE. Runs by default in dynamic class R. and also to manage multiple EJB instances. After EJBs have been written, they must be stored in a container residing on an application server. The container then manages all threading and client-interactions with the EJBs, and co-ordinate connection- and instance pooling.

control interval (CI)

A fixed-length area of disk storage where VSE/VSAM stores records and distributes free space. It is the unit of information that VSE/VSAM transfers to or from disk storage. For FBA it must be an integral multiple to be defined at cluster definition, of the block size.

control program

A program to schedule and supervise the running of programs in a system.

conversational monitor system (CMS)

A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities and operates under the control of z/VM.

count-key-data (CKD) device

A disk device that store data in the record format: count field, key field, data field. The count field contains, among others, the address of the record in the format: cylinder, head (track), record number, and the length of the data field. The key field, if present, contains the record's key or search argument. CKD disk space is allocated by tracks and cylinders. Contrast with *FBA disk device. See also extended count-key-data device.*

cross-partition communication control

A facility that enables VSE subsystems and user programs to communicate with each other; for example, with VSE/POWER.

cryptographic token

Usually referred to simply as a *token*, this is a device, which provides an interface for performing cryptographic functions like generating digital signatures or encrypting data.

cryptography

1. A method for protecting information by transforming it (encrypting it) into an unreadable format, called ciphertext. Only users who possess a secret key can decipher (or decrypt) the message into plaintext.
2. The transformation of data to conceal its information content and to prevent its unauthorized use or undetected modification .

D

data block group

The smallest unit of space that can be allocated to a VSE/POWER job on the data file. This allocation is independent of any device characteristics.

data conversion descriptor file (DCDF)

With a DCDF, you can convert individual fields within a record during data transfer between a PC and its host. The DCDF defines the record fields of a particular file for both, the PC and the host environment.

data import

The process of reformatting data that was used under one operating system such that it can subsequently be used under a different operating system.

Data Interfile Transfer, Testing, and Operations (DITTO) utility

An IBM program that provides file-to-file services for card I/O, tape, and disk devices. The latest version is called DITTO/ESA for VSE.

Data Language/I (DL/I)

A database access language that is used with CICS.

data link

In SNA, the combination of the link connection and the link stations joining network noes, for example, a z/Architecture channel and its associated protocols. A link is both logical and physical.

data security

The protection of data against unauthorized disclosure, transfer, modification, or destruction, whether accidental or intentional .

data set header record

In VSE/POWER abbreviated as DSHR, alias NDH or DSH. An NJE control record either preceding output data or, in the middle of input data, indicating a change in the data format.

data space

A range of up to 2 gigabytes of contiguous virtual storage addresses that a program can directly manipulate through z/Architecture instructions. Unlike an address space, a data space can hold only user data; it does not contain shared areas, or programs. Instructions do not execute in a data space. Contrast with address space.

data terminal equipment (DTE)

In SNA, the part of a data station that serves a data source, data sink, or both.

database connector

Is a function introduced with z/VSE 5.1.1, which consists of a client and server part. The client provides an API (CBCLI) to be used by applications on z/VSE, the server on any Java capable platform connects a JDBC driver that is provided by the database. Both client and server communicate via TCP/IP.

Database 2 (Db2)

An IBM rational database management system.

Db2-based connector

Is a feature introduced with VSE/ESA 2.5, which includes a customized Db2 version, together with VSAM and DL/I functionality, to provide access to Db2, VSAM, and DL/I data, using Db2 Stored Procedures.

Db2 Runtime only Client edition

The Client Edition for z/VSE comes with some enhanced features and improved performance to integrate z/VSE and Linux on z Systems.

Db2 Stored Procedure

In the context of z/VSE, a Db2 Stored Procedure is a Language Environment (LE) program that accesses Db2 data. However, from VSE/ESA 2.5 onwards you can also access VSAM and DL/I data using a Db2 Stored Procedure. In this way, it is possible to exchange data between VSAM and Db2.

DBLK

Data block.

DCDF

Data conversion descriptor file.

deblocking

The process of making each record of a block available for processing.

dedicated (disk) device

A device that cannot be shared among users.

device address

1. The identification of an input/output device by its device number.
2. In data communication, the identification of any device to which data can be sent or from which data can be received.

device driving system (DDS)

A software system external to VSE/POWER, such as a CICS spooler or PSF, that writes spooled output to a destination device.

Device Support Facilities (DSF)

An IBM supplied system control program for performing operations on disk volumes so that they can be accessed by IBM and user programs. Examples of these operations are initializing a disk volume and assigning an alternative track.

device type code

The four- or five-digit code that is used for defining an I/O device to a computer system. See also ICKDSF

dialog

In an interactive system, a series of related inquiries and responses similar to a conversation between two people. For z/VSE, a set of panels that can be used to complete a specific task; for example, defining a file.

dialog manager

The program component of z/VSE that provides for ease of communication between user and system.

digital signature

In computer security, encrypted data, which is appended to or part of a message, that enables a recipient to prove the identity of the sender.

Digital Signature Algorithm (DSA)

The Digital Signature Algorithm is the US government-defined standard for digital signatures. The DSA digital signature is a pair of large numbers, computed using a set of rules (that is, the DSA) and a set of parameters such that the identity of the signatory and integrity of the data can be verified. The DSA provides the capability to generate and verify signatures.

directory

In z/VSE the index for the program libraries.

direct access

Accessing data on a storage device using their address and not their sequence. This is the typical access on disk devices as opposed to magnetic tapes. Contrast with *sequential access*.

disk operating system residence volume (DOSRES)

The disk volume on which the system sublibrary IJSYSRS.SYSLIB is located including the programs and procedures that are required for system startup.

disk sharing

An option that lets independent computer systems uses common data on shared disk devices.

disposition

A means of indicating to VSE/POWER how a job input or output entry is to be handled: according to its local disposition in the RDR/LST/PUN queue or its transmission disposition when residing in the XMT queue. A job might, for example, be deleted or kept after processing.

distribution tape

A magnetic tape that contains, for example, a preconfigured operating system like z/VSE. This tape is shipped to the customer for program installation.

DITTO/ESA for VSE

Data Interfile Transfer, Testing, and Operations utility. An IBM program that provides file-to-file services for disk, tape, and card devices.

DSF

Device Support Facilities.

DSH (R)

Data set header record.

dummy device

A device address with no real I/O device behind it. Input and output for that device address are spooled on disk.

duplex

Pertaining to communication in which data can be sent and received at the same time.

DU-AL (dispatchable unit - access list)

The access list that is associated with a z/VSE main task or subtask. A program uses the DU-AL associated with its task and the PASN-AL associated with its partition. See also “PASN-AL (primary address space number - access list)” on page 250.

dynamic class table

Defines the characteristics of dynamic partitions.

dynamic partition

A partition that is created and activated on an 'as needed' basis that does not use fixed static allocations. After processing, the occupied space is released. Dynamic partitions are grouped by class, and jobs are scheduled by class. Contrast with *static partition*.

dynamic space reclamation

A librarian function that provides for space that is freed by the deletion of a library member to become reusable automatically.

E

ECI

See “CICS ECI” on page 233.

emulation

The use of programming techniques and special machine features that permit a computer system to execute programs that are written for another system or for the use of I/O devices different from those that are available.

emulation program (EP)

An IBM control program that allows a channel-attached 3705 or 3725 communication controller to emulate the functions of an IBM 2701 Data Adapter Unit, or an IBM 2703 Transmission Control.

end user

1. A person who makes use of an application program.
2. In SNA, the ultimate source or destination of user data flowing through an SNA network. Might be an application program or a terminal operator.

Enterprise Java Bean

An EJB is a distributed bean. "Distributed" means, that one part of an EJB runs inside the JVM of a web application server, while the other part runs inside the JVM of a web browser. An EJB either represents one data row in a database (entity bean), or a connection to a remote database (session bean). Normally, both types of an EJB work together. This allows to represent and access data in a standardized way in heterogeneous environments with relational and non-relational data. See also *JavaBean*.

entry-sequenced file

A VSE/VSAM file whose records are loaded without respect to their contents and whose relative byte addresses cannot change. Records are retrieved and stored by addressed access, and new records are added to the end of the file.

Environmental Record Editing and Printing (EREP) program

A z/VSE base program that makes the data that is contained in the system record file available for further analysis.

EPI

See *CICS EPI.*

ESCON Channel (Enterprise Systems Connection Channel)

A serial channel, using fiber optic cabling, that provides a high-speed connection between host and control units for I/O devices. It complies with the ESA/390 and IBM Z I/O Interface until z114. The zEC12 processors do not support ESCON channels.

exit routine

1. Either of two types of routines: installation exit routines or user exit routines. Synonymous with exit program.
2. See *user exit routine*.

extended addressability

The ability of a program to use 31 bit or 64 bit virtual storage in its address space or outside the address space.

extended recovery facility (XRF)

In z/VSE, a feature of CICS that provides for enhanced availability of CICS by offering one CICS system as a backup of another.

External Security Manager (ESM)

A priced vendor product that can provide extended functionality and flexibility that is compared to that of the Basic Security Manager (BSM), which is part of z/VSE.

F

FASTCOPY

See "VSE/Fast Copy" on page 261.

fast copy data set program (VSE/Fast Copy)

See "VSE/Fast Copy" on page 261.

fast service upgrade (FSU)

A service function of z/VSE for the installation of a refresh release without regenerating control information such as library control tables.

FAT-DASD

A subtype of Large DASD, it supports a device with more than 4369 cylinders (64 K tracks) up to 64 K cylinders.

FCOPY

See *VSE/Fast Copy.*

fence

A separation of one or more components or elements from the remainder of a processor complex. The separation is by logical boundaries. It allows simultaneous user operations and maintenance procedures.

fetch

1. To locate and load a quantity of data from storage.

2. To bring a program phase into virtual storage from a sublibrary and pass control to this phase.
3. The name of the macro instruction (FETCH) used to accomplish 2. See also *loader.*

Fibre Channel Protocol (FCP)

A combination of hardware and software conforming to the Fibre Channel standards and allowing system and peripheral connections via FICON and FICON Express feature cards on IBM zSeries processors. In z/VSE, zSeries FCP is employed to access industry-standard SCSI disk devices.

fragmentation (of storage)

Inability to allocate unused sections (fragments) of storage in the real or virtual address range of virtual storage.

FSU

Fast service upgrade.

FULIST (FUnction LIST)

A type of selection panel that displays a set of files and/or functions for the choice of the user.

G

generation

See *macro generation.*

generation feature

An IBM licensed program order option that is used to tailer the object code of a program to user requirements.

GETVIS space

Storage space within partition or the shared virtual area, available for dynamic allocation to programs.

guest system

A data processing system that runs under control of another (host) system. On the mainframe z/VSE can run as a guest of z/VM.

H

hard wait

The condition of a processor when all operations are suspended. System recovery from a hard wait is impossible without performing a new system startup.

hash function

A hash function is a transformation that takes a variable-size input and returns a fixed-size string, which is called the hash value. In cryptography, the hash functions should have some additional properties:

- The hash function should be easy to compute.
- The hash function is one way; that is, it is impossible to calculate the 'inverse' function.

- The hash function is collision-free; that is, it is impossible that different input leads to the same hash value.

hash value

The fixed-sized string resulting after applying a *hash function* to a text.

High-Level Assembler for VSE

A programming language providing enhanced assembler programming support. It is a base program of z/VSE.

home interface

Provides the methods to instantiate a new EJB object, introspect an EJB, and remove an EJB instantiation., as for the remote interface is needed because the deployment tool generates the implementation class. Every Session bean's home interface must supply at least one *create()* method.

host mode

In this operating mode, a PC can access a VSE host. For programmable workstation (PWS) functions, the Move Utilities of VSE can be used.

host system

The controlling or highest level system in a data communication configuration.

host transfer file (HTF)

Used by the Workstation File Transfer Support of z/VSE as an intermediate storage area for files that are sent to and from IBM personal computers.

HTTP Session

In the context of z/VSE, identifies the web-browser client that calls a servlet (in other words, identifies the connection between the client and the middle-tier platform).

I

ICCF

See *VSE/ICCF.*

ICKDSF (Device Support Facilities)

A z/VSE base program that supports the installation, use, and maintenance of IBM disk devices.

include function

Retrieves a library member for inclusion in program input.

index

1. A table that is used to locate records in an indexed sequential data set or on indexed file.
2. In, an ordered collection of pairs, each consisting of a key and a pointer, used by to sequence and locate the records of a key-sequenced data set or file; it is organized in levels of index records. See also *alternate index.*

input/output control system (IOCS)

A group of IBM supplied routines that handle the transfer of data between main storage and auxiliary storage devices.

integrated communication adapter (ICA)

The part of a processor where multiple lines can be connected.

integrated console

In z/VSE, the service processor console available on IBM Z that operates as the z/VSE system console. The integrated console is typically used during IPL and for recovery purposes when no other console is available.

Interactive Computing and Control Facility (ICCF)

An IBM licensed program that serves as interface, on a time-slice basis, to authorized users of terminals that are linked to the system's processor.

interactive partition

An area of virtual storage for the purpose of processing a job that was submitted interactively via VSE/ICCF.

Interactive User Communication Vehicle (IUCV)

Programming support available in a VSE supervisor for operation under z/VM. The support allows users to communicate with other users or with CP in the same way they would with a non-preferred guest.

intermediate storage

Any storage device that is used to hold data temporarily before it is processed.

IOCS

Input/output control system.

IPL

Initial program load.

irrecoverable error

An error for which recovery is impossible without the use of recovery techniques external to the computer program or run.

IUCV

Interactive User Communication Vehicle.

J

JAR

Is a platform-independent file format that aggregates many files into one. Multiple applets and their requisite components (.class files, images, and sounds) can be bundled in a JAR file, and then downloaded to a web browser using a single HTTP transaction (much improving the download speed). The JAR format also supports compression, which reduces the files size (and further improves the

download speed). The compression algorithm that is used is fully compatible with the ZIP algorithm. The owner of an applet can also digitally sign individual entries in a JAR file to authenticate their origin.

Java application

A Java program that runs inside the JVM of your web browser. The program's code resides on a local hard disk or on the LAN. Java applications might be large programs using graphical interfaces. Java applications have unlimited access to all your local resources.

Java bytecode

Bytecode is created when a file containing Java source language statements is compiled. The compiled Java code or "bytecode" is similar to any program module or file that is ready to be executed (run on a computer so that instructions are performed one at a time). However, the instructions in the bytecode are really instructions to the *Java Virtual Machine*. Instead of being interpreted one instruction at a time, bytecode is instead recompiled for each operating-system platform using a just-in-time (JIT) compiler. Usually, this enables the Java program to run faster. Bytecode is contained in binary files that have the suffix**.CLASS**

Java servlet

See *servlet.*

JHR

Job header record.

job accounting interface

A function that accumulates accounting information for each job step, to be used for charging the users of the system, for planning new applications, and for supervising system operation more efficiently.

job accounting table

An area in the supervisor where accounting information is accumulated for the user.

job catalog

A catalog made available for a job by means of the file name IJSYSUC in the respective DLBL statement.

job entry control language (JECL)

A control language that allows the programmer to specify how VSE/POWER should handle a job.

job step

In 1 of a group of related programs complete with the JCL statements necessary for a particular run. Every job step is identified in the job stream by an EXEC statement under one JOB statement for the whole job.

job trailer record (JTR)

As VSE/POWER parameter JTR, alias NJT. An NJE control record terminating a job entry in the input or output queue and providing accounting information.

K

key

In VSE/VSAM, one or several characters that are taken from a certain field (key field) in data records for identification and sequence of index entries or of the records themselves.

key sequence

The collating sequence either of records themselves or of their keys in the index or both. The key sequence is alphanumeric.

key-sequenced file

A VSE/VSAM file whose records are loaded in key sequence and controlled by an index. Records are retrieved and stored by keyed access or by addressed access, and new records are inserted in the file in key sequence.

KSDS

Key-sequenced data sets. See *key-sequenced file.*

L

label

1. An identification record for a tape, disk, or diskette volume or for a file on such a volume.
2. In assembly language programming, a named instruction that is generally used for branching.

label information area

An area on a disk to store label information that is read from job control statements or commands. Synonymous with *label area*.

Language Environment for z/VSE

An IBM software product that is the implementation of Language Environment on the VSE platform.

language translator

A general term for any assembler, compiler, or other routine that accepts statements in one language and produces equivalent statements in another language.

Large DASD

A DASD device that

1. Has a capacity exceeding 64 K tracks and
2. Does not have VSAM space created prior to VSE/ESA 2.6 that is owned by a catalog.

LE/VSE

Short form of Language Environment for z/VSE.

librarian

The set of programs that maintains, services, and organizes the system and private libraries.

library block

A block of data that is stored in a sublibrary.

library directory

The index that enables the system to locate a certain sublibrary of the accessed library.

library member

The smallest unit of a data that can be stored in and retrieved from a sublibrary.

line commands

In VSE/ICCF, special commands to change the declaration of individual lines on your screen. You can copy, move, or delete a line declaration, for example.

linkage editor

A program that is used to create a phase (executable code) from one or more independently translated object modules, from one or more existing phases, or from both. In creating the phase, the linkage editor resolves cross-references among the modules and phases available as input. The program can catalog the newly built phases.

linkage stack

An area of protected storage that the system gives to a program to save status information for a branch and stack or a stacking program call.

link station

In SNA, the combination of hardware and software that allows a node to attach to and provide control for a link.

loader

A routine, commonly a computer program, that reads data or a program into processor storage. See also *relocating loader*.

local shared resources (LSR)

A VSE/VSAM option that is activated by three extra macros to share control blocks among files.

lock file

In a shared disk environment under VSE, a system file on disk that is used by the sharing systems to control their access to shared data.

logical partition

In LPAR mode, a subset of the server unit hardware that is defined to support the operation of a system control program.

logical record

A user record, normally pertaining to a single subject and processed by data management as a unit. Contrast with *physical* record, which may be larger or smaller.

logical unit (LU)

1. A name that is used in programming to represent an I/O device address. *physical unit (PU), system services control point (SSCP), primary logical unit (PLU), and secondary logical unit (SLU).*
2. In SNA, a port through which a user accesses the SNA network,
   a. To communicate with another user and
   b. To access the functions of the SSCP. An LU can support at least two sessions. One with an SSCP and one with another LU and might be capable of supporting many sessions with other LUs.

logical unit name

In programming, a name that is used to represent the address of an input/output unit.

logical unit 6.2

A SNA/SDLC protocol for communication between programs in a distributed processing environment. LU 6.2 is characterized by

1. A peer relationship between session partners,
2. Efficient utilization of a session for multiple transactions,
3. Comprehensive end-to-end error processing, and
4. A generic Application Programming Interface (API) consisting of structured verbs that are mapped into a product implementation.

logons interpret interpret routine

In VTAM, an installation exit routine, which is associated with an interpret table entry, that translates logon information. It also verifies the logon.

LPAR mode

Logically partitioned mode. The CP mode that is available on the Configuration (CONFIG) frame when the PR/SM feature is installed. LPAR mode allows the operator to allocate the hardware resources of the processor unit among several logical partitions.

M

macro definition

A set of statements and instructions that defines the name of, format of, and conditions for generating a sequence of assembler statements and machine instructions from a single source statement.

macro expansion

See *macro generation*

macro generation

An assembler operation by which a macro instruction gets replaced in the program by the statements of its definition. It takes place before assembly. Synonymous with *macro expansion*.

macro (instruction)

1. In assembler programming, a user-invented assembler statement that causes the assembler to process a set of statements that are defined previously in the macro definition.

2. A sequence of VSE/ICCF commands that are defined to cause a sequence of certain actions to be performed in response to one request.

maintain system history program (MSHP)

A program that is used for automating and controlling various installation, tailoring, and service activities for a VSE system.

main task

The main program within a partition in a multiprogramming environment.

master console

In z/VSE, one or more consoles that receive all system messages, except for those that are directed to one particular console. Contrast this with the *user* console, which receives only those messages that are specifically directed to it, for example messages that are issued from a job that was submitted with the request to echo its messages to that console. The operator of a master console can reply to all outstanding messages and enter all system commands.

maximum (max) CA

A unit of allocation equivalent to the maximum control area size on a count-key-data or fixed-block device. On a CKD device, the max CA is equal to one cylinder.

memory object

Chunk of virtual storage that is allocated above the bar (2 GB) to be created with the IARV64 macro.

message

In VSE, a communication that is sent from a program to the operator or user. It can appear on a console, a display terminal or on a printout.

MSHP

See maintain system history program.

multitasking

Concurrent running of one main task and one or several subtasks in the same partition.

MVS

Multiple Virtual Storage. Implies MVS/390, MVS/XA, MVS/ESA, and the MVS element of the z/OS (OS/390) operating system.

N

NetView

A z/VSE optional program that is used to monitor a network, manage it, and diagnose its problems.

network address

In SNA, an address, consisting of subarea and element fields, that identifies a link, link station, or NAU. Subarea nodes use network addresses; peripheral nodes use local addresses. The boundary function in the subarea node to which a peripheral node is attached transforms local addresses to network addresses and vice versa. See also *network name*.

network addressable unit (NAU)

In SNA, a logical unit, a physical unit, or a system services control point. It is the origin or the destination of information that is transmitted by the path control network. Each NAU has a network address that represents it to the path control network. See also *network name, network address*.

Network Control Program (NCP)

An IBM licensed program that provides communication controller support for single-domain, multiple-domain, and interconnected network capability. Its full name is ACF/NCP.

network definition table (NDT)

In VSE/POWER networking, the table where every node in the network is listed.

network name

1. In SNA, the symbolic identifier by which users refer to a NAU, link, or link station. See also *network address*.
2. In a multiple-domain network, the name of the APPL statement defining a VTAM application program. This is its network name, which must be unique across domains.

node

1. In SNA, an end point of a link or junction common to several links in a network. Nodes can be distributed to host processors, communication controllers, cluster controllers, or terminals. Nodes can vary in routing and other functional capabilities.
2. In VTAM, a point in a network that is defined by a symbolic name. Synonymous with *network node*. See *major node and minor node*.

node type

In SNA, a designation of a node according to the protocols it supports and the network addressable units (NAUs) it can contain.

O

object module (program)

A program unit that is the output of an assembler or compiler and is input to a linkage editor.

online application program

An interactive program that is used at display stations. When active, it waits for data. Once input arrives, it processes it and send a response to the display station or to another device.

operator command

A statement to a control program, issued via a console or terminal. It causes the control program to provide requested information, alter normal operations, initiate new operations, or end existing operations.

optional licensed program

An IBM licensed program that a user can install on VSE by way of available installation-assist support.

output parameter text block (OPTB)

in VSE/POWER's spool-access support, information that is contained in an output queue record if a * $$ LST or * $$ PUN statement includes any user-defined keywords that have been defined for autostart.

P

page data set (PDS)

One or more extents of disk storage in which pages are stored when they are not needed in processor storage.

page fixing

Marking a page so that it is held in processor storage until explicitly released. Until then, it cannot be paged out.

page I/O

Page-in and page-out operations.

page pool

The set of page frames available for paging virtual-mode programs.

panel

The complete set of information that is shown in a single display on terminal screen. Scrolling back and forth through panels like turning manual pages. See also *selection panel*.

partition balancing

A z/VSE facility that allows the user to specify that two or more or all partitions of the system should receive about the same amount of time on the processor.

PASN-AL (primary address space number - access list)

The access list that is associated with a partition. A program uses the PASN-AL associated with its partition and the DU-AL associated with its task (work unit). See also *DU-AL*.

Each partition has its own unique PASN-AL. All programs running in this partition can access data spaces through the PASN-AL. Thus a program can create a data space, add an entry for it in the PASN-AL, and obtain the ALET that indexes the entry. By passing the ALET to other programs in the partition, the program can share the data space with other programs running in the same partition.

PDS

Page data sets.

phase

The smallest complete unit of executable code that can be loaded into virtual storage.

physical record

The amount of data that is transferred to or from auxiliary storage. Synonymous with *block*.

PNET

Programming support available with VSE/POWER; it provides for the transmission of selected jobs, operator commands, messages, and program output between the nodes of a network.

POWER

See *VSE/POWER*.

pregenerated operating system

An operating system such as z/VSE that is shipped by IBM mainly in object code. IBM defines such key characteristics as the size of the main control program, the organization, and size of libraries, and required system areas on disk. The customer does not have to generate an operating system.

preventive service

The installation of one or more PTFs on a VSE system to avoid the occurrence of anticipated problems.

primary address space

In z/VSE, the address space where a partition is executed. A program in primary mode fetches data from the primary address space.

primary library

A VSE library owned and directly accessible by a certain terminal user.

printer/keyboard mode

Refers to 1050 or 3215 console mode (device dependent).

Print Services Facility (PSF)/VSE

An access method that provides support for the advanced function printers.

private area

The virtual space between the shared area (24 bit) and shared area (31 bit), where (private) partitions are allocated. Its maximum size can be defined during IPL. See also *shared area*.

private memory object

Memory object (chunk of virtual storage) that is allocated above the 2 GB line (bar) only accessible by the partition that created it.

private partition

Any of the system's partitions that are not defined as shared. See also *shared partition*.

production library

1. In a pre-generated operating system (or product), the program library that contains the object code for this system (or product).
2. A library that contains data that is needed for normal processing. Contrast with *test library*.

programmer logical unit

A logical unit available primarily for user-written programs. See also *logical unit name*.

program temporary fix (PTF)

A solution or by-pass of one or more problems that are documented in APARs. PTFs are distributed to IBM customers for preventive service to a current release of a program.

PSF/VSE

Print Services Facility/VSE.

PTF

See *Program temporary fix*.

Q

Queue Control Area (QCA)

In VSE/POWER, an area of the data file, which might contain:

- Extended checkpoint information
- Control information for a shared environment.

queue file

A direct-access file that is maintained by VSE/POWER that holds control information for the spooling of job input and job output.

R

random processing

The treatment of data without respect to its location on disk storage, and in an arbitrary sequence that is governed by the input against which it is to be processed.

real address area

In z/VSE, processor storage to be accessed with dynamic address translation (DAT) off

real address space

The address space whose addresses map one-to-one to the addresses in processor storage.

real mode

In VSE, a processing mode in which a program might not be paged. Contrast with *virtual mode*.

recovery management support (RMS)

System routines that gather information about hardware failures and that initiate a retry of an operation that failed because of processor, I/O device, or channel errors.

refresh release

An upgraded VSE system with the latest level of maintenance for a release.

relative-record file

A VSE/VSAM file whose records are loaded into fixed-length slots and accessed by the relative-record numbers of these slots.

release upgrade

Use of the FSU functions to install a new release of z/VSE.

relocatable module

A library member of the type object. It consists of one or more control sections cataloged as one member.

relocating loader

A function that modifies addresses of a phase, if necessary, and loads the phase for running into the partition that is selected by the user.

remote interface

In the context of z/VSE, the remote interface allows a client to make method calls to an EJB although the EJB is on a remote z/VSE host. The container uses the remote interface to create client-side stubs and server-side proxy objects to handle incoming method calls from a client to an EJB.

remote procedure call (RPC)

1. A facility that a client uses to request the execution of a procedure call from a server. This facility includes a library of procedures and an external data representation.
2. A client request to service provider in another node.

residency mode (RMODE)

A program attribute that refers to the location where a program is expected to reside in virtual storage. RMODE 24 indicates that the program must reside in the 24-bit addressable area (below 16 megabytes), RMODE ANY indicates that the program can reside anywhere in 31-bit addressable storage (above or below 16 megabytes).

REXX/VSE

A general-purpose programming language, which is particularly suitable for command procedures, rapid batch program development, prototyping, and personal utilities.

RMS

Recovery management support.

RPG II

A commercially oriented programming language that is specifically designed for writing application programs that are intended for business data processing.

S

SAM ESDS file

A SAM file that is managed in VSE/VSAM space, so it can be accessed by both SAM and VSE/VSAM macros.

SCP

System control programming.

SDL

System directory list.

search chain

The order in which chained sublibraries are searched for the retrieval of a certain library member of a specified type.

second-level directory

A table in the SVA containing the highest phase names that are found on the directory tracks of the system sublibrary.

Secure Sockets Layer (SSL)

A security protocol that allows the client to authenticate the server and all data and requests to be encrypted. SSL was developed by Netscape Communications Corp. and RSA Data Security, Inc..

segmentation

In VSE/POWER, a facility that breaks list or punch output of a program into segments so that printing or punching can start before this program has finished generating such output.

selection panel

A displayed list of items from which a user can make a selection. Synonymous with *menu*.

sense

Determine, on request or automatically, the status or the characteristics of a certain I/O or communication device.

sequential access method (SAM)

A data access method that writes to and reads from an I/O device record after record (or block after block). On request, the support performs device control operations such as line spacing or page ejects on a printer or skip some tape marks on a tape drive.

service node

Within the VSE unattended node support, a processor that is used to install and test a master VSE system, which is copied for distribution to the unattended nodes. Also, program fixes are first applied at the service node and then sent to the unattended nodes.

service program

A computer program that performs function in support of the system. See with *utility program*.

service refresh

A form of service containing the current version of all software. Also referred to as a *system refresh*.

service unit

One or more PTFs on disk or tape (cartridge).

shared area

In z/VSE, shared areas (24 bit) contain the Supervisor areas and SVA (24 bit) and shared areas (31 bit) the SVA (31 bit). Shared areas (24 bit) are at the beginning of the address space (below 16 MB), shared area (31 bit) at the end (below 2 GB).

shared disk option

An option that lets independent computer systems use common data on shared disk devices.

shared memory objects

Chunks of virtual storage allocated above the 2 GB line (bar), that can be shared among partitions.

shared partition

In z/VSE, a partition that is allocated for a program (VSE/POWER, for example) that provides services and communicates with programs in other partitions of the system's virtual address spaces. In most cases shared partitions are no longer required.

shared spooling

A function that permits the VSE/POWER account file, data file, and queue file to be shared among several computer systems with VSE/POWER.

shared virtual area (SVA)

In z/VSE, a high address area that contains a list system directory list (SDL) of frequently used phases, resident programs that are shared between partitions, and an area for system support.

SIT (System Initialization Table)

A table in CICS that contains data used the system initialization process. In particular, the SIT can identify (by suffix characters) the version of CICS system control programs and CICS tables that you have specified and that are to be loaded.

skeleton

A set of control statements, instructions, or both, that requires user-specific information to be inserted before it can be submitted for processing.

socksified

See *socks-enabled*.

Socks-enabled

Pertaining to TCP/IP software, or to a specific TCP/IP application, that understands the *socks protocol*. "Socksified" is a slang term for socks-enabled.

socks protocol

A protocol that enables an application in a secure network to communicate through a firewall via a *socks server*.

socks server

A circuit-level gateway that provides a secure one-way connection through a firewall to server applications in a nonsecure network.

source member

A library member containing source statements in any of the programming languages that are supported by VSE.

split

To double a specific unit of storage space (CI or CA) dynamically when the specified minimum of free space gets used up by new records.

spooling

The use of disk storage as buffer storage to reduce processing delays when transferring data between peripheral equipment and the processor of a computer. In z/VSE, this is done under the control of VSE/POWER.

Spool Access Protection

An optional feature of VSE/POWER that restricts individual spool file entry access to user IDs that have been authenticated by having performed a security logon.

spool file

1. A file that contains output data that is saved for later processing.
2. One of three VSE/POWER files on disk: queue file, data file, and account file.

SSL

See Secure Sockets Layer.

stacked tape

An IBM supplied product-shipment tape containing the code of several licensed programs.

standard label

A fixed-format record that identifies a volume of data such as a tape reel or a file that is part of a volume of data.

stand-alone program

A program that runs independently of (not controlled by) the VSE system.

startup

The process of performing IPL of the operating system and of getting all subsystems and applications programs ready for operation.

start option

In VTAM, a user-specified or IBM specified option that determines conditions for the time a VTAM system is operating. Start options can be predefined or specified when VTAM is started.

static partition

A partition, which is defined at IPL time and occupying a defined amount of virtual storage that remains constant. See also *dynamic partition*.

storage director

An independent component of a storage control unit; it performs all of the functions of a storage control unit and thus provides one access path to the disk devices that are attached to it. A storage control unit has two storage directors.

storage fragmentation

Inability to allocate unused sections (fragments) of storage in the real or virtual address range of virtual storage.

suballocated file

A VSE/VSAM file that occupies a portion of an already defined data space. The data space might contain other files. See also *unique file*.

sublibrary

In VSE, a subdivision of a library. Members can only be accessed in a sublibrary.

sublibrary directory

An index for the system to locate a member in the accessed sublibrary.

submit

A VSE/POWER function that passes a job to the system for processing.

SVA

See shared virtual area.

Synchronous DataLink Control (SDLC)

A discipline for managing synchronous, code-transparent, serial-by-bit information transfer over a link connection. Transmission exchanges might be duplex or half-duplex over switched or non-switched links. The configuration of the link connection might be point-to-point, multipoint, or loop.

SYSRES

See system residence volume.

system control programming (SCP)

IBM supplied, non-licensed program fundamental to the operation of a system or to its service or both.

system directory list (SDL)

A list containing directory entries of frequently used phases and of all phases resident in the SVA. The list resides in the SVA.

system file

In z/VSE, a file that is used by the operating system, for example, the hardcopy file, the recorder file, the page data set.

System Initialization Table (SIT)

A table in CICS that contains data that is used by the system initialization process. In particular, the SIT can identify (by suffix characters) the version of CICS system control programs and CICS tables that you have specified and that are to be loaded.

system recorder file

The file that is used to record hardware reliability data. Synonymous with *recorder file*.

system refresh

See *service refresh*.

system refresh release

See *refresh release*.

system residence file (SYSRES)

The z/VSE system sublibrary IJSYSRS.SYSLIB that contains the operating system. It is stored on the system residence volume DORSES.

system residence volume (SYSRES)

The disk volume on which the system sublibrary is stored and from which the hardware retrieves the initial program load routine for system startup.

system sublibrary

The sublibrary that contains the operating system. It is stored on the system residence volume (SYSRES).

T

task management

The functions of a control program that control the use, by tasks, of the processor and other resources (except for input/output devices).

time event scheduling support

In VSE/POWER, the time event scheduling support offers the possibility to schedule jobs for processing in a partition at a predefined time once repetitively. The time event scheduling operands of the * $$ JOB statement are used to specify the wanted scheduling time.

TLS

See Transport Layer Security.

track group

In VSE/POWER, the basic organizational unit of a file for CKD devices.

track hold

A function that protects a track that is being updated by one program from being accessed by another program.

transaction

1. In a batch or remote batch entry, a job or job step. 2. In CICS TS, one or more application programs that can be used by a display station operator. A given transaction can be used concurrently from one or more display stations. The execution of a transaction for a certain operator is also referred to as a task.
2. A given task can relate only to one operator.

transient area

An area within the control program that is used to provide high-priority system services on demand.

Transport Layer Security

The newest SSL cryptographic protocol. It provides additional strength to privacy and data integrity.

Turbo Dispatcher

A facility of z/VSE that allows to use multiprocessor systems (also called CEC: Central Electronic Complexes). Each CPU within such a CEC has accesses to be shared virtual areas of z/VSE: supervisor, shared areas (24 bit), and shared areas (31 bit). The CPUs have equal rights, which means that any CPU might receive interrupts and work units are not dedicated to any specific CPU.

U

UCB

Universal character set buffer.

universal character set buffer (UCB)

A buffer to hold UCS information.

UCS

Universal character set.

user console

In z/VSE, a console that receives only those system messages that are specifically directed to it. These are, for example, messages that are issued from a job that was submitted with the request to echo its messages to that console. Contrast with *master console*.

user exit

A programming service that is provided by an IBM software product that can be requested during the execution of an application program for the service of transferring control back to the application program upon the later occurrence of a user-specified event.

V

variable-length relative-record data set (VRDS)

A relative-record data set with variable-length records. See also *relative-record data set*.

variable-length relative-record file

A VSE/VSAM relative-record file with variable-length records. See also *relative-record file*.

VIO

See virtual I/O area.

virtual address

An address that refers to a location in virtual storage. It is translated by the system to a processor storage address when the information stored at the virtual address is to be used.

virtual addressability extension (VAE)

A storage management support that allows to use multiple virtual address spaces.

virtual address space

A subdivision of the virtual address area (virtual storage) available to the user for the allocation of private, nonshared partitions.

virtual disk

A range of up to 2 gigabytes of contiguous virtual storage addresses that a program can use as workspace. Although the virtual disk exists in storage, it appears as a real FBA disk device to the user program. All I/O operations that are directed to a virtual disk are intercepted and the data to be written to, or read from, the disk is moved to or from a data space.

Like a data space, a virtual disk can hold only user data; it does not contain shared areas, system data, or programs. Unlike an address space or a data space, data is not directly addressable on a virtual disk. To manipulate data on a virtual disk, the program must perform I/O operations.

Starting with z/VSE 5.2, a virtual disk may be defined in a shared memory object.

virtual I/O area (VIO)

An extension of the page data set; used by the system as intermediate storage, primarily for control data.

virtual mode

The operating mode of a program, where the virtual storage of the program can be paged, if not enough processor (real) storage is available to back the virtual storage.

virtual partition

In VSE, a division of the dynamic area of virtual storage.

virtual storage

Addressable space image for the user from which instructions and data are mapped into processor storage locations.

virtual tape

In z/VSE, a virtual tape is a file (or data set) containing a tape image. You can read from or write to a virtual tape in the same way as if it were a physical tape. A virtual tape can be:

- A VSE/VSAM ESDS file on the z/VSE local system.
- A remote file on the server side; for example, a Linux, UNIX, or Windows file. To access such a remote virtual tape, a TCP/IP connection is required between z/VSE and the remote system.

volume ID

The volume serial number, which is a number in a volume label that is assigned when a volume is prepared for use by the system.

VRDS

Variable-length relative-record data sets. See *variable-length relative record file*.

VSAM

See *VSE/VSAM*.

VSE (Virtual Storage Extended)

A system that consists of a basic operating system and any IBM supplied and user-written programs that are required to meet the data processing needs of a user. VSE and hardware it controls form a complete computing system. Its current version is called z/VSE.

VSE/Advanced Functions

A program that provides basic system control and includes the supervisor and system programs such as the Librarian and the Linkage Editor.

VSE Connector Server

Is the host part of the VSE JavaBeans, and is started using the job STARTVCS, which is placed in the reader queue during installation of z/VSE. Runs by default in dynamic class R.

VSE/DITTO (VSE/Data Interfile Transfer, Testing, and Operations Utility)

An IBM licensed program that provides file-to-file services for disk, tape, and card devices.

VSE/ESA (Virtual Storage Extended/Enterprise Systems Architecture)

The predecessor system of z/VSE.

VSE/Fast Copy

A utility program for fast copy data operations from disk to disk and dump/restore operations via an intermediate dump file on magnetic tape or disk.

VSE/FCOPY (VSE/Fast Copy Data Set program)

An IBM licensed program for fast copy data operations from disk to disk and dump/restore operations via an intermediate dump file on magnetic tape or disk. There is also a stand-alone version: the FASTCOPY utility.

VSE/ICCF (VSE/Interactive Computing and Control Facility)

An IBM licensed program that serves as interface, on a time-slice basis, to authorized users of terminals that are linked to the system's processor.

VSE/ICCF library

A file that is composed of smaller files (libraries) including system and user data, which can be accessed under the control of VSE/ICCF.

VSE JavaBeans

Are JavaBeans that allow access to all VSE-based file systems (VSE/VSAM, Librarian, and VSE/ICCF), submit jobs, and access the z/VSE operator console. The class library is contained in the *VSEConnector.jar* archive. See also *JavaBeans*.

VSE library

A collection of programs in various forms and storage dumps stored on disk. The form of a program is indicated by its member type such as source code, object module, phase, or procedure. A VSE library consists of at least one sublibrary, which can contain any type of member.

VSE/POWER

An IBM licensed program that is primarily used to spool input and output. The program's networking functions enable a VSE system to exchange files with or run jobs on another remote processor.

VSE/VSAM (VSE/Virtual Storage Access Method)

An IBM access method for direct or sequential processing of fixed and variable length records on disk devices.

VSE/VSAM catalog

A file containing extensive file and volume information that VSE/VSAM requires to locate files, to allocate and deallocate storage space, to verify the authorization of a program or an operator to gain access to a file, and to accumulate use statistics for files.

VSE/VSAM managed space

A user-defined space on disk that is placed under the control of VSE/VSAM.

W

wait for run subqueue

In VSE/POWER, a subqueue of the reader queue with dispatchable jobs ordered in execution start time sequence.

wait state

The condition of a processor when all operations are suspended. System recovery from a hard wait is impossible without performing a new system startup. See *hard wait*.

Workstation File Transfer Support

Enables the exchange of data between IBM Personal Computers (PCs) linked to a z/VSE host system where the data is kept in intermediate storage. PC users can retrieve that data and work with it independently of z/VSE.

work file

A file that is used for temporary storage of data being processed.

Numerics

24-bit addressing

Provides addressability for address spaces up to 16 megabytes.

31-bit addressing

Provides addressability for address spaces up to 2 gigabytes.

64-bit addressing

Provides addressability for address spaces up to 2 gigabytes and above.

Index

Numerics

2540 punch/read codes 85
3505 card device
  file, close the 84
  optical-mark read 81
  read column eliminate 81
3525 card device
  associated files on 79
  card-print control codes 85
  card-print function 84
  device control 85
  device-control restriction 85
  file, close the 84
  print file, associated 88
  read column eliminate 81
3800 table reference number 20
4248 printer
  horizontal copy 91
  output to 91
  print buffer 91

A

abnormal-end
  exit 112
  job step 113
  subtask 120
access control, ANSI tape 77
access method
  criteria for choosing 21
  direct 189
  indexed sequential 24, 35
  languages, supporting 24
  support summary 22
  virtual storage (VSE/VSAM) 23
  VSE/VSAM 23
access time 6
access type 6
accessibility 227
accessing librarian member data 140
accessing library objects (members) 139
activate a subtask 118
add a record
  coding for 205
  programming requirements 192
address
  home 7
  I/O device 6
  note a 52
address reference 194
advance page-in 102
AFTER operand 205
alternate
  blocks/tracks 7
  record transfer 43
  tape switching
alternate *(continued)*
  tape switching *(continued)*
    control before 76
    end of volume 68, 69
    PIOCS file 217
area definition
  combined file 81
  I/O for SAM
    address of 43
ASCII file
  access to 77
  block prefix 19
  block size 65
  file label for 25
  label processing 70
  spanned records 12
  tape records 19
ASOCFLE operand
  3525 print file 88
  purpose 79
ASPL macro 105
assembly
  examples of 182
  modifications for separate 184
  printed output, sample program 184
  program using IOCS 181
  separate vs. joint 186
assign logical unit
  coding example 106
  macro for 105
  parameter list 105
ASSIGN macro 105
associated file
  close a, on 3525 84
  GET-CNTRL-PUT sequence 79
  print, IBM 3525 88
  read-print 79
  read-punch 79
  read-punch-print 79
attach a subtask 118
ATTACH macro 118

B

BLKSIZE operand 43
BLKSIZE=MAX specification 43
block
  boundaries 12
  descriptor 12
  FBA 15
  length of 43
  of records 12
  prefix, ASCII 19
  tape 65
block size
  indication 12
  SAM file 43

block size *(continued)*
tape file 65
blocked records
concept 12
DAM processing 189
SAM input 47
SAM output 48
selective processing by SAM 49
blocking factor 12
boundary
data block 12
partition 102
buffer
control interval 64
forms control (FCB) 117
print 91
universal character set (UCS) 89

C

CALL macro 115
called program 113
calling program 113
CANCEL macro 113, 120
capacity record
layout of 192
use of 197
capital letters output 89
card file
2540 card feeding 84
3525 print function 84
close 3505/3525 file 84
control codes 85
define a 79
end of 84
error handling 84
card punch control codes 85
card read control codes 85
card records 20
carriage control 20
catalog a program 184
CATALOG command 181
CCB (command control block) macro 215
CDDELETE macro 99
CDLOAD macro 99
CDMOD macro 79
chain of sublibraries
processing sequence of 143
searching for 141
channel program
CKD disk 217
command chaining in 215
command-chain retry in 217
contents of 214
data chaining in 217
example 217
execution of 215
FBA disk 217
indirect data addressing in 217
rotational position sensing 217
user-written 211
CHAP macro 118
character arrangement table 20
character set 20
CHECK macro
work file 51
checkpoint restart
on disk 211
on tape 217
CIDF (control interval definition field) 19
clear a track 205
clear disk utility 192
Clear Disk utility 198
clear print buffer 91
CLOSE/CLOSER macro
3505 file 84
3525 file 84
card I/O 84
direct processing 210
disk sequential 63
PIOCS file 217
sequential file 55
closing library members 140
CNTRL macro
card devices 85
direct processing 207
printer control 89
seek operation 207
sequential processing 55
tape
block skip 75
buffer synchronization 76
file skip 75
logical-record skip 75
rewind the 75
summary 74
combined file 81
command chaining
CKD disk 215
FBA disk 217
retry 217
command control block (CCB) macro 215
communication
area 103
job step to job step 103
job to job 103
region 103
task to task 120
communication path
defining a 150
compression
of data
description 167
symbols 166
COMRG macro 103
console buffering (PIOCS) 217
console file
define a 93
record length 20
write with reply 93
continuation character 2
control block
cross-partition communication (XPCCB) 148
control character
card output 87
printer 88
control codes
2540 punch/read 85

control codes *(continued)*
3505 card read 85
3525 card punch 85
3525 card-print 85
printer 89
control function macro
assign I/O unit 105
dump request 132
exit-routine linkage 110
forms-control buffer load 117
groups of 99
I/O unit, assign/release 105
in reenterable program 221
job end
abnormal 112
normal 113
linkage 115
multitasking 118
program communication 103
program linkage 113
program loading 99
release I/O unit 105
resource sharing 125
system information request 134
timer services 108
types of 99
virtual storage control 100
control information request macro 134
control interval
data transfer by 64
size of 15
spanned records 19
conversion algorithm 197
count area (CKD or ECKD disk) 16
count-key-data (CKD and ECKD)
count area 16
disk channel program 217
key area 16
record format 16
cross-partition communication
abnormal end processing 165
clearing a request 161
control block (XPCCB) 148
data transmission 152
defining a communication path 150
disconnecting from a communication link 163
identifying an application 149
macros 148
MAPXPCCB macro 148
receiving data 158
replying to a request 160
sending and receiving data 152
terminating XPCC usage 164
transferring data 152
XPCC macro 148
XPCCB macro 148
CSRCMPSC macro 165, 178, 179
cylinder concept 7

D

DAM processing
add a record 192
advantage of 189
DAM processing *(continued)*
clear a track 205
error handling 207
file creation 192
file reorganization 198
free space 201
imperative macros for 191
LBRET macro 192
load a file 198
locate
data 193
free space 197
logic module for 189
open file for 192
read a record
completion 207
reference by ID 204
reference by key 204
seek for 207
record with key
fixed length 198
read a 198
variable length 198
record without key
coding requirements 199
randomizing algorithm 199
request macros for 197
summary 23
track reference 194
versus sequential 22
WAITF macro 207
write a record
add to a DAM file 205
by overwrite 205
completion 207
end-of-file 205
reference by ID 205
reference by key 205
seek for 207
spanned 205
to a DAM file, overview 205
undefined 205
variable-length 205
with clear track function 205
with verification 205
write records 205
DASD record protection (track hold) 126
data
receiving of 158
sending and receiving of 152
transfer 152
transmission 152
data area (CKD or ECKD disk) 16
data card, OMR 82
data chaining (PIOCS) 217
data format
block of records 12
card I/O 20
CKD or ECKD disk 7, 16
console I/O 20
control interval 15
device-dependent 16
FBA disk 7, 19
spanned records 12

data format *(continued)*
variable-length records 12
data management
access method 21
device characteristics 6
direct access method (DAM) 189
format of data 12
I/O control system (IOCS) 34
labels 25
overview 5
physical IOCS (PIOCS) 211
sequential access method (SAM) 57
storage of data 11
volume of data 7
data spaces
dumps of 132
data transfer rate 6
deadlock 122
declarative macro
assembly of 181
catalog 184
CDMOD 79
DIMOD 95
DTFCD 79
DTFCN 93
DTFDA 189
DTFDI 95
DTFMT 65
DTFPH 211
DTFPR 88
DTFSD 57
example 34
link-edit a 184
PRMOD 88
purpose of 34
request macro relation 34
request macros for 55
SAM processing 41
separate assembly 184
separate vs. joint assembly 186
define the lock (DTL) macro 125
deleting a phase (CDDELETE) 99
deleting library members 141
DEQ macro 118, 122
descriptor card 81
descriptor, block/record
spanned records 12
variable-length records 12
detach a subtask 120
DETACH macro 120
device address
CKD and ECKD disk 7
physical 6
device capacity 6
device characteristics
access time 6
access type 6
addressing 6
direct access 6
disk address 7
disk data formats 7
disk volume 7
requesting 133
serial access 6
device characteristics *(continued)*
storage capacity 6
tape volume 10
transfer rate 6
volume of data 7
device control
direct access 207
seek disk 207
sequential processing
3525 card print 85
card I/O control characters 87
card-stacker selection 85
print control characters 88
printed output 88
summary 55
tape file 74
device independent file
end of file 97
error handling 96
full-track support 43
process a 95
record size 95
restrictions 95
device sharing 36
Device Support Facilities 198
dictionary
compression 166
entries 168
expansion 166
direct access method (DAM) 189
directory list 100
disability 227
disk
addressing, CKD and ECKD 7
cylinder 7
data formats 7
direct access 189
extent 7
file label 26
home address 7
record zero 7
split cylinder extent 7
track 7
volume
initialization 7
label 7
layout 27
work file on 50
disk file
direct access 189
sequential access 57
disk file label
end-of-file processing 59
end-of-volume processing 59
OPEN processing 58
user standard 59
DOM macro 135
DTF table 36
DTF to module and program relation 182
DTFDA macro 189
DTL (define the lock) macro 125
DUMP macro 113, 120, 132
dump of storage 132

dynamic allocation of storage 103

E

ECB (event control block) 118
end a subtask 120
end of extent
    PIOCS file 212
    SAM file 57
end of file
    card I/O 84
    device-independent file 97
    disk sequential 59
    specification in DTFxx 45
end of volume
    disk 59
    forced, tape 69
    multivolume file, tape 71
    PIOCS file
        SYSLST/SYSPCH on tape 216
        user data 216
ENQ macro 118, 122
EODAD librarian exit 144
EOJ macro 113
EOXPTR operand 57, 212
ERET macro
    device-independent file 96
    in error routine 61
    in WLRERR routine 62
    purpose 44
ERRAD librarian exit
    librarian 144
ERRBYTE operand 207
ERREXT support 72
ERROPT operand 44
error handling
    sequential-disk file
        exit for 61
        I/O error 63
        options 61
        wrong-length record 62
    tape file
        exit for 72
        options for 72
        SAM functions 74
        wrong-length record 74
error indicator
    DAM file 207
error option (ERROPT)
    action for 44
    disk sequential 61
    tape file 72
    tape, summary 72
event control block (ECB) 118
EXCP (execute channel program) macro 215
execution mode 102
EXIT macro 110
exit routine
    CLOSE interface 76
    disk error, SAM 61
    disk label, PIOCS 213
    disk label, SAM 58
    end of extent
    end of extent *(continued)*
        PIOCS file 212
    end of extent, SAM 57
    end of file 45
    end-of-volume interface 76
    EODAD (librarian) 144
    ERRAD (librarian) 144
    error option, SAM 44
    extent exit 192
    extent processing
        PIOCS file 212
    linkage
        abnormal end 112
        conditions for 110
        interval timer 110
        operator communication 113
        program check 112
        register usage 110
        set up a 110
    NOTFND (librarian) 144
    OPEN interface 76
    PIOCS file, label 213
    PIOCS, end of extent 212
    print overflow 88
    return from (SAM) 44
    tape error 72
    tape label 65
    tape management 76
expansion
    of data
        description 167
extent
    checking of by SAM
        input 57
        output 57
    disk 7
    end of, SAM processing 47
    exit routine 192
    file/volume relationship 11
    PIOCS, check a 213
    split cylinder 7
extent fields 29
EXTRACT macro 134, 216
extracting
    partition information 134
    system information 134

F

fast load (of a phase) 100
FCB (forms-control buffer) 117
FEOV macro 69, 71, 216
FEOVD macro 63
FETCH macro 99
file
    checkpointed 211
    close a
        macro for 55
        PIOCS file 217
        SAM processing 55
        tape 55
    combined 81
    define a

file *(continued)*
define a *(continued)*
direct processing 189
PIOCS access 211
SAM processing 41
device independent 95
direct access 189
disk, sequential 57
end of, SAM processing 47
extent/volume relationship 11
name of 34
open a
disk sequential 57
SAM processing 45
tape file 45
unit record 45
optical mark read 81
organization 20
physical IOCS 211
punch card 79
read/write by SAM 47
records of 12
reopen, sequential disk 63
reorganization 198
tape 65
type of, SAM 42
unit record 79
unlabeled 68
update 60
work 50
file definition
end of extent 57
record format 42
SAM processing
logic module 189
record type 189
fix a program page 101
fixed block architecture
block on 15
disk channel program 217
record format 19
space optimization 15
fixed-length record
DAM processing
with key 198
without key 199
SAM processing
output 48
short block 49
TRUNC macro 49
unblocked 12
floating point registers 2
FOLD option 89
force
end of volume
disk sequential 63
PIOCS file 216
tape 68
page-in 102
format
descriptor card 81
generate 221
label 27
macro-operand, mixed 2
forms-control buffer (FCB) 20, 117
free a program page 101
FREE macro 126
free space
control interval 15
direct access file 197
FREEVIS macro 103
full-track support 43
FUNC operand
3525-print file 88
associated file
read-print 79
read-punch 79
read-punch-print 79
summary 79
purpose 79

G

gap, interblock 10
GENDTL macro 125
generate
command control block 215
I/O request block 215
local directory list 100
lock control block 125
generate format 221
GENIORB (generate IORB) macro 215
GENL macro 100
GET (get record) macro
associated file 79
at end of file 47
blocked records 47
combined file 81
selective read 49
spanned records 47
unblocked records 47
update file 60
work file 47
GET request for library member 140
GET/CNTRL/PUT sequence 79
GETIME macro 108
GETVCE macro 133
GETVIS area 99
GETVIS macro 103

H

hints for programming
2540 card feeding 84
3505 file, close a 84
3525 card-print function 84
3525 file, close a 84
PIOCS file
alternate tape switching 217
checkpoint on disk 211
checkpoint on tape 217
CKD disk channel program 217
command-chain retry 217
console buffering 217
data chaining 217
FBA disk channel program 217
indirect data addressing 217

hints for programming *(continued)*
PIOCS file *(continued)*
LIOCS functions 217
rotational position sensing 217
track-hold function 127
home address 7
homepage, z/VSE xv
horizontal-copy function 91

I

I/O (input/output)
register 43
request block (IORB) macro 215
I/O error
device-independent I/O 96
disk sequential 61
printed output 92
tape 72
ID reference
read record by 204
write record by 205
IDAW (indirect data addressing) 217
identification
of communication user 149
identifier (ID) reference 196
IDLOC operand 204
imperative (request) macros 34
indexed sequential access method
restrictions 35
summary 24
indirect data addressing 217
input file
define a 42
disk sequential
end of volume 63
open 57
process labels 58
update records of 60
disk sequential, open 57
optical mark read 81
PIOCS file
checkpoint bypass 217
disk open 213
disk-label processing 213
extent processing 213
read a record 214
tape-label processing 213
sequential 47
stacker selection 85
tape
CLOSE if unopened 74
end of volume 71
label processing 67
nonstandard labels 67
unlabeled 68
input/output control system
direct processing 189
logical versus physical 39
macros, overview 34
physical 211
primary functions 5
sequential processing (SAM) 41
interblock gap 10, 32
Internet address, z/VSE homepage xv
intertask communication 120
interval timer
example of use 109
exit 110
purpose 109
unexpired interval 109
wait on 109
IOAREA1 operand 43
IOAREA2 operand. 43
IOCS macros
declarative 34
module generation (xxMOD) 37
overview 34
request 34
IOCS modules, assembling and cataloging 181
IORB (I/O request block) macro 215
IOREG operand 43

J

JDUMP macro 113, 120, 132
job end
abnormal 112
normal 113
job step
cancel a 132
communication 103
end of 113
job-to-job communication 103
JOBCOM macro 103
joint vs. separate assembly 186

K

key area (CKD or ECKD disk) 16
key reference
read record by 204
write record by 205
keyword operand 2

L

label
disk file 26
formats on disk 27
tape file 26
types 25
volume 26
label processing
concepts 25
direct access file 192
disk sequential
CLOSE processing 59
end of volume 59
OPEN processing 58
requirements 58
return to SAM 60
user standard 59
PIOCS file
disk input, multivolume 213
disk output, multivolume 212

label processing *(continued)*
PIOCS file *(continued)*
disk output, single volume 212
disk, check a 213
disk, write a 213
nonstandard 213
tape, check a 213
tape, write a 213
user labels, disk file 213
tape file
input 67
output 65
requirements 65
return to SAM 70
languages, programming 24
layout of system, extracting 134
LBRET macro
direct access 192
disk sequential file 60
PIOCS file
disk label, check a 213
disk, write a 213
extent, check a 213
nonstandard label 213
tape label, check a 213
tape, write a 213
tape file 70
LDCB (librarian data control block) 140
LFCB (load forms-control buffer) macro
coding example 117
librarian data control block (LDCB) 140
librarian exits
EODAD (end-of-member data) 144
ERRAD (error) 144
NOTFND (not found) 144
librarian member data, accessing 140
librarian member data, updating 140
library access
for application programs 139
Library Access Service
control block 140
exits 144
for accessing library objects 139
functions 141
LDCB 140
LIBRDCB macro 140
LIBRM macro 140
reason codes 142
record formats 143
record I/O 140
register usage 144
request sequence 147
return codes 142
library chain
dropping a 141
establishing a 141
processing sequence of 143
searching for 141
library members
accessing 139
closing 140, 141
deleting 141
locking 141
NOTE/POINT processing of 140, 141
library members *(continued)*
opening 140, 141
qualified 140
renaming 141
retrieving 140, 141
searching for 141
specifying 140
unlocking 141
writing 141
library objects 139
library, searching for 141
LIBRDCB (library control block) macro 139
LIBRM (library) macro 139
line skipping/spacing 88
link-edit 184
linkage macros 115
linkage register 113
LIOCS functions for PIOCS 217
LOAD macro 99
loading
forms-control buffer 117
program/phase 99
local directory list 100
locate data 193
lock control block 125
LOCK macro 125
lock request count 125
locking a library member 141
LOCKOPT (resource sharing) 125
logic module
assembly of 181
catalog 184
direct access 189
generation macros 34
generation of 37
link-edit a 184
preassembled 37
reentrant 128
relation of to DTF/program 182
separate assembly 184
separate vs. joint assembly 186
share a 128
standard name 38
subsetting/supersetting of 38
supply the name of 39
logical IOCS
access methods of 35
versus physical 39
logical unit name
assign a 36, 105
example of use 36
permitted 36
PIOCS file 211
purpose of 36
release a 105
lowercase control 89

M

macro
continuation of 2
control function 2, 99
control program 1
cross-partition communication 148

macro *(continued)*
declarative 34
format of 2
generate format (MFG) 221
IOCS overview 34
operands 2
PIOCS 217
purpose 1
registers for 2
request 34
types of 1
magnetic tape records 19
MAPBDY macro 102, 134
MAPBDYVR macro 102, 134
MAPEXTR macro 102, 134
mapping system layout (MAPEXTR macro) 134
MAPSSID macro 134
MAPSYSP macro 134
MAPXPCCB macro 148
MFG (macro format generate) 221
mixed format (macro operands) 2
MODDTL macro 125
mode, program run 102
multifile volume 11
multiple-track search
read, ID reference 204
read, key reference 204
write, ID reference 205
write, key reference 205
multitasking
common user-exit 110
communication, task to task 120
end subtask 120
event control block 118
I/O considerations 118
priority change 118
resource protection 122
resource sharing 125
Sample Program 129
save area for 118
start a subtask 118
task to task communication 120
test attachment of 118
multivolume file 11
multivolume processing
disk sequential
end of volume 63
open input 57
open output 57
PIOCS file open
input 213
output 212
tape 71
MVCOM macro 103

N

name of file 34
non-standard tape label
checking of 67, 70
writing of 65, 70
NOTE macro 52
NOTE macro for library access 140
NOTFND librarian exit 144

O

OMR (optical-mark read) mode 81
OMR coding example 82
open a file
for access by PIOCS
disk input 213
disk labels 211
disk output 212
inactive 211
tape labels 211
OPEN/OPENR macro
by self-relocating program 45
coding example 45
direct access 192
disk sequential
input 57
output 57
PIOCS file 211
SAM processing 45
opening a library member 140
operands, macro
keyword 2
mixed (keyword/positional) 2
positional 2
operator-communication exit 113
optical mark read (OMR)
close file 84
data card 82
descriptor card 81
mode of operation 81
process data 81
weak mark 82
output file
define a 42
direct access (DAM) 192
disk sequential
end of 63
end of volume 63
open 57
process labels 58
PIOCS access
disk open 212
disk-label processing 213
tape-label processing 213
write a record 214
tape
CLOSE if unopened 74
end of volume 71
extension of 71
label processing 65
unlabeled 69
overflow
area
access record in 199
cylinder vs. independent 198
descriptor record 201
organization 201
print 88
routine 197
overwriting a record 205

P

padding (of data) 12
page
  fault (interrupt) 102
  print, layout 20
PAGEIN macro 102
parameter
  for routing messages 135
  list, ASSIGN macro 105
  register 2, 113
parity error 63
partition
  boundary 102
  communication region 103
  dump of 132
PDUMP macro 132
PFIX macro 101
PFREE macro 101
phase, load a 99
physical IOCS (PIOCS) 39, 211
PIOCS file
  channel program for 217
  check user-standard label
    disk 213
    tape 213
  close a 217
  end of volume 216
  extent processing for 213
  hints for programming 217
  label processing for 213
  macro relationship 217
  open a 211
  process a 214
  programming for 211
  transmission-information bits 215
  user-option bits 215
  write nonstandard label 213
  write user-standard label
    disk 213
    tape 213
POINT macro for library access 140
POINTR macro (for work files) 52
POINTS macro 52
POINTW macro 52
positional operand 2
POST macro 120
preassembled logic module 37
prime data area 199
print buffer, IBM 4248 91
print-control buffer
  LFCB macro 117
  load a 20
  purpose of 20
print-control code
  card 85
  printer 89
printer file
  4248 printer 91
  associated, IBM 3525 88
  device control
    CNTRL macro 89
    control characters 88
    methods 88
printer file *(continued)*
  error handling 92
  FOLD/UNFOLD option 89
  forms-control buffer 117
  horizontal copy 91
  lowercase control 89
  page (print) overflow 88
  process a 88
  records for 20
  selective tape listing 89
  universal character set (UCS) 89
  uppercase control 89
priority change 118
PRMOD macro 88
process data
  direct access file 197
  disk, sequential 57
  PIOCS file 214
  sequential (by SAM) 41
  tape file 65
processing-priority change 118
processor storage
  fix/free a page 101
  transfer rate 6
program
  check user exit 112
  communication 103
  example, IOCS 182
  link-edit a 184
  linkage 113
  load a 99
  reenterable 221
  run mode inquiry 102
  to DTF/module relation 182
program page
  fix a 101
  free a 101
  page in a 102
  release a 102
programming languages 24
PRTOV macro 88
punch-read-feed feature 81
punched-card file 79
purge print buffer 91
PUT (a record) request 48
PUT request for library member 140
PUTR macro 93

Q

qualified library member 140

R

randomize
  cylinder address 198
  methods, summary 203
  record address 197
  track address 198
RCB (resource control block) macro 122
RCE (read-column-eliminate) mode 81
RDF (record definition field) 19
READ macro

READ macro *(continued)*
DAM file 204
work file 51
read-column-eliminate
close file 84
descriptor card 81
mode of operation 81
process data 81
read-only module 128
receiving of data 158
RECFORM operand 42
record
ASCII tape 19
block of 12
blocking 16
capacity 192
card I/O 20
CKD or ECKD disk 16
console 20
DAM file 189
DAM processing 189
deblocking 16
definition field (RDF) 19
descriptor 12
FBA disk 19
fields of 12
fixed length 12
free space for 201
identifier (ID) 194
length of 42
logical 12
overflow-area descriptor 201
overwrite a 205
padding of 12
physical 12
prime data to overflow relation 199
print output 20
read a
direct 204
PIOCS file 214
sequential 47
SAM formats 42
selective processing 49
spanned 12
synonym 197
tape 19
truncation of 12
undefined 12
variable length 12
with a key area 198
without a key area 199
write a
direct 205
PIOCS file 214
sequential 48
zero 192
record I/O
for accessing library members 140
record identifier (ID) 194
record key 193
record length
device independent file 95
SAM processing 42
record reference
record reference *(continued)*
by AFTER 205
by ID, read 204
by ID, write 205
by identifier (ID) 196
by key 196
by key, read 204
by key, write 205
record zero 7
RECSIZE operand 42
reenterable program
coding example 221
control-function macros in 221
register notation 221
S,address notation 221
reflective marker 69
register
conventions for use 2
floating point 2
I/O 43
linkage 2, 113
notation, example 221
save area 2, 113
relative track reference 194
release
block, short 49
I/O unit 105
page
FCEPGOUT macro 102
force out a 102
page-out 102
RELEASE macro 106
RELPAG macro 102
RELSE macro 49
renaming library members 141
reopen file
disk sequential 63
request macro
declarative macro relation 34
direct processing 191
I/O device relation 36
purpose 34
SAM processing 55
resource
control block (RCB) 122
dequeue a 122
enqueue a 122
protection 122
share a 122
share control 125
restrictions
VTOC size 7
work file 50
RETURN macro 115
rotational position sensing (RPS) 217
routine, linkage 113
routing codes 135
RPS (rotational position sensing) 217
run mode 102
RUNMODE macro 102
RZERO reference 205

S

S,address notation 221
SAM processing
  associated file 79
  blocked records
    input 47
    output 48
    selective 49
  card device control 85
  card I/O 79
  combined file 81
  console file 93
  define file for 41
  device control 55
  device independent 95
  disk file 57
  end of extent 47, 57
  end of file 45, 47
  end of processing 55
  end of volume
    disk sequential 63
    output 48
    SAM processing 47
  error handling 44
  file close 55
  file type 42
  for FBA disk 64
  general 47
  GET macro 47
  I/O area 43
  I/O area length 43
  I/O register 43
  non-data device operation 55
  open file 45
  optical mark read data 81
  optical-reader file 81
  printer file 88
  PUT macro 48
  read-column-eliminate data 81
  record format 42
  record length 42
  record-format definition 42
  reopen file 63
  selective 49
  system file 95
  tape 65
  unblocked records
    input 47
    output 48
  undefined records
    input 47
    output 48
  update file 60
  variable-length record
    output 48
    TRUNC macro 49
  versus direct 22
  work area 44
  work area for 44
  work file
    open a 50
    purpose of 50
    selective processing 52
SAM processing *(continued)*
  work file *(continued)*
    sequential processing 51
save area
  multitasking 118
  register 113
SAVE macro 115
SDL (system directory list) 100
SDUMP macro 132
SDUMPX macro 132
SECTVAL macro 216
SEEK operand 207
seek operation
  command for 217
seek separation feature 217
segment-sequence error 63
selective processing
  records 49
  work file 52
selective tape listing 89
self-relocating program
  open a 45, 192
sending and receiving data 152
sense information 215
SEOF macro 63
SEOV macro 216
separate vs. joint assembly 186
sequential access method
  define a file to 41
  disk file 57
  processing 41
  summary 23
  tape file 65
  unit record file 79
SETIME macro 109
shared resource
  across partitions 125
  control of
    coding example 122
    macros for 125
    resource control block 122
  define a 125
  I/O module 128
  within a partition 122
shared virtual area (SVA)
  load a phase from 100
  system directory list 100
shared Virtual Area (SVA) 99
sharing a device 36
single volume, PIOCS open
  input 213
  output 212
skipping of lines 88
space one line 88
space optimization, FBA 15
space/skip control
  CNTRL macro 89
  control characters 88
  methods 88
spanned record
  CKD or ECKD Disk 16
  DAM processing 189
  FBA disk 19
  layout of 12

spanned record *(continued)*
SAM, input 47
SAM, output 48
segment-sequence error 63
tape 19
tape file extension 71
split-cylinder extent 7
SRCHM operand
read, ID reference 204
read, key reference 204
stacker selection
card I/O 20, 85
standard labels
disk file 58
identification 25
location of 28
PIOCS file
disk, check a 213
disk, write a 213
nonstandard 213
tape, check a 213
tape, write a 213
SAM processing 45
tape file
ASCII file 77
extended 72
input 67
output 65
tape files 65
standard module names 38
start a subtask 118
status indicator
ERRBYTE field (DAM) 207
storage
allocation, dynamic 103
capacity 6
dump of 132
streaming mode 10
STXIT macro 110
sublibrary chain
processing sequence of 143
searching for 141
sublibrary, searching for 141
subsetting a logic module 38
SUBSID macro 134
subtask
attach a 118
communication 120
detach a 132
end a
abnormal 112
normal 120
event control block 118
priority change 118
resource protection 122
resource sharing 125
save area for 118
start a 118
terminate a 120
verify attachment of 118
supersetting a logic module 38
supervisor
dump of 132
SVA (shared virtual area) 99
synonym record 197
system
directory list (SDL) 100
information request macro 134
layout (MAPEXTR macro) 134
logical units 36
system-mapping (MAPEXTR) macro 134

T

table reference number 20
table, DTF 36
tape file
ASCII, access to 77
assign logical unit to 105
block size 65
end of volume 71
extension of 71
file label 26
initialization 10
labels for 65
non-data operation 74
nonstandard labels 10
process a 65
records of 19
release assigned logical unit 105
tape file label
ASCII file 70
checking of 67
writing of 65
unlabeled 10, 68
user interface 76
volume
characteristics 10
end of 71
label 10
layout 32
tape listing, selective 89
tape positioning
input 68
output 69
tape-management exit 76
tapemark
behind labels 213
for file positioning 68
forced end-of-volume 216
purpose of 32
system end-of-volume 216
TECB (timer event control block) macro 109
terminate a subtask 120
TIC command 217
timer event control block (TECB) 109
timer services
interval 109
time-of-day 108
TOD (time-of-day) clock 108
track balance, retrieving (GETVCE) 133
track capacity, retrieving (GETVCE) 134
track hold
coding example 127
hints for programming 127
prerequisites 126
scope of support 126
track reference (DAM)

track reference (DAM) *(continued)*
actual address 194
format of 194
full address (mbbcchhr) 194
record-ID format 194
relative address 194
Transfer in Channel (TIC) command 217
transfer rate 6
transmission
of data 152
transmission information 215
TRUNC macro (SAM) 49
truncation (of data) 12
TTIMER macro 109
type (of file) specification 42
TYPEFLE operand 42

U

UCB (universal character-set buffer) 20
UCS feature 89
unblocked records
DAM file 189
SAM input 47
SAM output 48
undefined record
DAM file 189
SAM input processing 47
SAM output processing 48
structure of 12
unexpired time 109
UNFOLD option 89
unit of transfer 12
unit record file
card I/O 79
console I/O 93
devices for 79
printed output 88
universal character set 89
universal character-set buffer 20
unlabeled file
input 68
output 69
process a 68
UNLOCK macro 125
unlocking a library member 141
update file
coding example 60
define a 42
process a 60
punched cards 81
updating librarian member data 140
uppercase control 89
user-option bits 215

V

VARBLD operand 43
variable-length records
blocked 12
DAM processing
with key 198
without key 199
variable-length records *(continued)*
layout 12
SAM processing
I/O area 43
I/O register 43
output processing 48
TRUNC macro 49
virtual storage access method (VSE/VSAM) 23
virtual storage control
dynamic storage allocation 103
fix/free a page 101
page control 102
partition boundary 102
run mode inquiry 102
summary 100
virtual storage dump 132
VOL1 label 25
volume
disk
alternate blocks/tracks 7
cylinder concept (CKD and ECKD) 7
data format 7
extents 7
initialization 7
layout 27
relative-block concept (FBA) 7
end of 63, 71
file/extent relationship 11
forced end of 69
input, PIOCS open 213
label
disk 7
identification 25
placement of 26
store a 26
tape 10
output, PIOCS open 212
table of contents 26
tape 10
volume characteristics
requesting 133
volume initialization
disk 7
tape 10
volume layout
disk 27
examples 29
tape
multifile volume 32
multivolume file 32
nonstandard labels 32
single-volume file 32
unlabeled 32
volume table of contents (VTOC)
capacity of 28
contents of 27
ISAM consideration 28
location of 28
size of 7

W

WAIT macro 109, 118, 215
WAITF macro

WAITF macro *(continued)*
DAM processing 207
WAITM macro 118
weak mark 82
WLRERR operand 44
work area
define a (SAM) 44
example for using 44, 45
work file
CHECK macro 51
define a 42
delete after use 54
example
define 50
selective processing 52
labels for 50
NOTE macro 52
on tape
block size 65
tape mark in 50
open a 50
POINTR macro 52
POINTS macro 52
POINTW macro 52
position a
at a record 52
beginning of file 52
behind a record 52
purpose of 50
READ macro 51
retain after use 54
selective processing 52
sequential processing 51
WRITE macro 51
WORKA operand 44
WRITE macro
AFTER reference 205
clear a track 205
ID reference 205
key reference 205
RZERO reference 205
to a DAM file 205
work file 51
write verification 205
wrong-length error
device-independent file 96
disk sequential 62
tape 74
WTO macro 135
WTOR macro 135

X

XPCC macro 148
XPCCB (cross-partition communication control block) 148
XPCCB macro 148
XTNTXIT operand
DAM file 192
input file 213
output file 212
xxMOD macro 34

Product Number: 5686-VS6

SC34-2709-01