Transaction Processing Facility

Database Reference

Version 4 Release 1

SH31-0143-14

Transaction Processing Facility

Database Reference

Version 4 Release 1

SH31-0143-14

Note!

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

Fifteenth Edition (June 2002)

This is a major revision of, and obsoletes, SH31-0143-13 and all associated technical newsletters.

This edition applies to Version 4 Release 1 Modification Level 0 of IBM Transaction Processing Facility, program number 5748-T14, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

IBM welcomes your comments. Address your comments to:

IBM Corporation
TPF Systems Information Development
Mail Station P923
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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.

Contents

Figures

Tables

Notices

References in this book to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service in this book is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 830A
Mail Drop P131
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Any pointers in this book to non-IBM Web sites are provided for convenience only and do not in any way serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this book or accessed through an IBM Web site that is mentioned in this book.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

3090
ECKD
ES/3090
ES/9000
ESCON
IBM
MQSeries
MVS/ESA
OS/390
Processor Resource/Systems Manager
S/390.

About This Book

This book contains information about the Transaction Processing Facility (TPF) system and the planning, programming, and operations required to access data in an applications and operating environment.

In this book, abbreviations are often used instead of spelled-out terms. Every term is spelled out at first mention followed by the all-caps abbreviation enclosed in parentheses; for example, Systems Network Architecture (SNA). Abbreviations are defined again at various intervals throughout the book. In addition, the majority of abbreviations and their definitions are listed in the master glossary in the *TPF Library Guide*.

Before You Begin

You should have a basic understanding of the TPF system, as well as the direct access storage devices (DASD) and access methods supported by the TPF system. See *TPF Concepts and Structures* for an overview of the TPF system. See *TPF Migration Guide: Program Update Tapes* for information about supported hardware and software for the TPF system.

You should also be familiar with MVS because these data processing techniques are used in the TPF system generation and offline processing.

Who Should Read This Book

This book is intended for system programmers responsible for the planning and implementing of applications running under the TPF system.

How This Book is Organized

This book is organized into the following major parts:

- Part 1, “Database Organization” on page 1
- Part 2, “Caching Support” on page 33
- Part 3, “Utilities” on page 53
- Part 4, “Coupling Facility Support” on page 221.

Conventions Used in the TPF Library

The TPF library uses the following conventions:

Conventions	Examples of Usage
italic	Used for important words and phrases. For example: A *database* is a collection of data. Used to represent variable information. For example: Enter **ZFRST STATUS MODULE** *mod*, where *mod* is the module for which you want status.
bold	Used to represent text that you type. For example: Enter **ZNALS HELP** to obtain help information for the ZNALS command. Used to represent variable information in C language. For example: **level**

Conventions	**Examples of Usage**

`monospaced`	Used for messages and information that displays on a screen. For example:
	`PROCESSING COMPLETED`
	Used for C language functions. For example:
	`maskc`
	Used for examples. For example:
	`maskc(MASKC_ENABLE, MASKC_IO);`

bold italic	Used for emphasis. For example: You ***must*** type this command exactly as shown.
Bold underscore	Used to indicate the default in a list of options. For example: **Keyword=OPTION1** | **DEFAULT**
Vertical bar |	Used to separate options in a list. (Also referred to as the OR symbol.) For example: **Keyword=Option1** | **Option2** **Note:** Sometimes the vertical bar is used as a *pipe* (which allows you to pass the output of one process as input to another process). The library information will clearly explain whenever the vertical bar is used for this reason.
CAPital LETters	Used to indicate valid abbreviations for keywords. For example: KEYWord=*option*

Scale	Used to indicate the column location of input. The scale begins at column position 1. The plus sign (+) represents increments of 5 and the numerals represent increments of 10 on the scale. The first plus sign (+) represents column position 5; numeral 1 shows column position 10; numeral 2 shows column position 20 and so on. The following example shows the required text and column position for the image clear card.
	`|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...`
	`LOADER   IMAGE CLEAR`
	Notes:
	1. The word LOADER must begin in column 1.
	2. The word IMAGE must begin in column 10.
	3. The word CLEAR must begin in column 16.

Related Information

A list of related information follows. For information on how to order or access any of this information, call your IBM representative.

IBM Transaction Processing Facility (TPF) 4.1 Books

- *TPF Application Programming*, SH31-0132
- *TPF C/C++ Language Support User's Guide*, SH31-0121
- *TPF Concepts and Structures*, GH31-0139
- *TPF General Macros*, SH31-0152
- *TPF Main Supervisor Reference*, SH31-0159
- *TPF Migration Guide: Program Update Tapes*, GH31-0187
- *TPF Operations*, SH31-0162
- *TPF System Generation*, SH31-0171
- *TPF System Installation Support Reference*, SH31-0149
- *TPF System Macros*, SH31-0151

- *TPF Transmission Control Protocol/Internet Protocol*, SH31-0120.

Miscellaneous IBM Books

- *ES/9000, ES/3090 Input/Output Configuration Program User's Guide and ESCON Channel-to-Channel Reference*, GC38-0097
- *MVS/ESA JCL Reference* (order the correct version and release for your installation)
- *OS/390 MVS Sysplex Services Guide*, GC28-1771
- *S/390 Processor Resource/Systems Manager Planning Guide*, GA22-7236
- *3880 Storage Control Record Cache RPQ Description*, GA32-0087
- *3880 Storage Control Record Cache RPQ Introduction*, GA32-0086
- *3990 Storage Control Introduction*, GA32-0098
- *3990 Storage Control Operations and Recovery Guide*, GA32-0253
- *3990 Storage Control Planning, Installation, and Storage Administration*, GA32-0100
- *3990 Transaction Processing Facility Support RPQs*, GA32-0134.

Online Information

- *Messages (Online)*
- *Messages (System Error and Offline)*.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TPF information, use one of the methods that follow. Make sure you include the title and number of the book, the version of your product and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send your comments electronically, do either of the following:
  - Go to http://www.ibm.com/tpf/pubs/tpfpubs.htm.

    There you will find a link to a feedback page where you can enter and submit comments.
  - Send your comments by e-mail to tpfid@us.ibm.com
- If you prefer to send your comments by mail, address your comments to:

  IBM Corporation
  TPF Systems Information Development
  Mail Station P923
  2455 South Road
  Poughkeepsie, NY 12601-5400
  USA
- If you prefer to send your comments by FAX, use this number:
  - United States and Canada: 1 + 845 + 432 + 9788
  - Other countries: (international code) + 845 + 432 +9788

Part 1. Database Organization

This part contains information about the organization of the database, including:

- File addressing
- General data sets and general files
- Pool support.

File Address Formats

The TPF system supports 3 basic logical file structures. They are:
- Online files
- General files
- General data sets.

Online files provide a common data repository for related online applications. These applications can provide independent function, but all online applications share the same data repository or online database.

General files and general data sets are usually related to some offline processing. Data is either produced online to be processed offline by MVS programs or data is produced offline for online processing. Both structures provide a data interface between the offline and online system components.

General Files

TPF general files are sequentially organized sets of data that are not MVS compatible. See “General Files” on page 17 and *TPF System Generation* for more information about general files.

General Data Sets

TPF general data sets are directly related to MVS data sets in the same module (MVS volume). These records can be processed online and offline.

Online Processing

The find and file type macros are used for online processing of TPF fixed and pool records, while the file data chain transfer (FDCTC) macro allows processing of records not restricted to TPF file record sizes.

Offline Processing

Offline, standard MVS facilities such as the GET macro are used to process the records of a TPF general data set. See “General Data Sets” on page 13 and *TPF System Generation* for more information about general data sets.

Online Database Addresses

The file address that the FACE program or the get file storage macros generates for use with find or file type macro requests exists in one of the following formats:
- FARF3 (file address reference format 3)
- FARF4 (file address reference format 4)
- FARF5 (file address reference format 5)
- FARF6 (file address reference format 6).

Figure 1 on page 4 describes these formats.

Note: In this chapter, the term FACE is used as a generic name for the FACE, FACS, FACZC, or FAC8C program.

Bit	FARF3 Fixed	FARF3 Pool	FARF4	FARF5	FARF6	General File RRN
00	0=fixed	1=pool	Universal Format Type	Universal Format Type	Space Reserved for IBM Use Only	General File Symbolic Module Number
01	Band Number	0=LT, 1=ST				
02		Ordinal Number				
03						
04						
05						
06			Variable-Size Format Type Indicator	Variable-Size Format Type Indicator		
07						
08					Universal Format Type	Relative Record Number
09						
10						
11						
12						
13	Ordinal Number					
14						
15						
16						
17						
18						
19						
20						
21						
22			Variable-Size Ordinal Number	Variable-Size Ordinal Number		
23						
24					Variable-Size Format Type Indicator (8-24 bits in size)	
25						
26						
27						
28		1				
29	0 = Simplex 1 = Duplex	0 = Simplex 1 = Duplex				1
30	1	1	0			0
31	0 = Small 1 = Large	0 = Small 1 = Large	0 = Small 1 = Large			0 = Small 1 = Large
• • • • 63					Variable-Size Ordinal Number (16-32 bits in size)	

Figure 1. File Address Formats

FARF Format

With FARF addressing, a database ordinal number is recalculated each time the find and file type macros find the format that represents a record type and ordinal number; the configuration dependencies are bound to channel programs that are based on their current assignments and held in the FACE table (FCTB). This means that imbedded addresses that use FARF format are less sensitive to file reorganizations because of the late binding of the referenced *imbedded* address to the actual *physical* address of the record.

Nevertheless, data must be moved to correspond to any new allocations. This is done by a TPF capture of online data using the directories and FCTB of the original allocation. This is then followed by a TPF restore using the directories and FCTB of the new allocation.

The first of the FARF formats is file address reference format 3 (FARF3) and is shown in Figure 1 on page 4. Based on the sizes of the band and ordinal number fields in a FARF3 fixed address, there are 2^{12} bands and 2^{16} ordinals. This yields an addressing range of as many as 2^{28} (268 435 456) fixed records. “Mapping FARF Addresses” explains the methods and reasons for assigning band numbers to fixed records.

There is no band field in a FARF3 pool address. Each pool record that is defined in the TPF system is related to a record type based on pool type, size, and duplication. The bits in the FARF3 pool address indicate which type the address belongs to. Each pool record type can have as many as 2^{26} records because 26 bits is the length of the ordinal number field.

The FARF4, FARF5, and FARF6 formats are also shown in Figure 1 on page 4. FARF4, FARF5, and FARF6 addresses make no distinction between fixed and pool addresses. With 2 bits of a FARF4 address delegated to control information, this provides an addressing space of 2^{30}. FARF5 has no control bits reserved in the address, so 2^{32} (4 294 967 296) addresses can be represented. FARF6 can represent 2^{56} addresses (72 057 594 037 927 936). All control information for FARF5 and FARF6 addresses is found in the FACE table. The dividing line between the format type indicator (FTI) field and the ordinal field in FARF4, FARF5, and FARF6 addresses is not in a fixed position. For each universal format type (UFT) value, a different FTI size can be related.

Note: The UFT is 6 bits for FARF4 and FARF5 (64 possible UFT values). For FARF6, the UFT is 16 bits (65 536 possible UFT values).

Once the FTI size is known, the FTI value, the ordinal size, and the ordinal value can be determined. “Mapping FARF Addresses” describes how these values relate to specific ordinal numbers of a record type.

Mapping FARF Addresses

During FACE table generation, the details of the specific address formats to use for each fixed and pool record type are assigned. For FARF3 fixed records, one band number needs to be assigned for each 64K (65 536) ordinal numbers. Because there is an upper limit on bands, there is an upper limit to the number of fixed record types that FARF3 can map. Consider an installation that insists on 4096 fixed record types; in this case each record type is restricted to 65 536 ordinal numbers. At the other extreme, consider an installation that insist on a single fixed record type; here, 4096 blocks of 65 536 ordinal numbers can be allocated to ***1*** record type.

In FARF4, FARF5, and FARF6, each UFT/FTI combination has a specific address capacity based on the size of the FTI associated with the UFT. Depending on the number of records needed for a record type, you must assign enough UFT/FTI combinations to map this number of records.

Assigning bands and UFT/FTIs to record types produces a mapping of each ordinal number in a record type to a specific FARF address.

FARF3 pools were left out of the preceding discussion because there is no FARF descriptor related to each pool ordinal (where a descriptor is a band or UFT/FTI). The control bits in the pool address are the only mapping from a record type to a pool address. By inspecting a FARF3 pool address the specific ordinal and the pool type it belongs to are revealed. The relationship between record types and their descriptors needs to be known to determine this level of information from any other FARF address. The disadvantage of FARF3 is the limit on addressing capacity because of the presence of control bits. This disadvantage decreases with FARF4 and does not exist with FARF5.

Regardless of the address format, generating a FARF address for a fixed or pool record is done identically. The record type of the record creates an index into the FACE table.

Note: Pool records have record type names and record type equate values that are assigned to them by the FACE table generator.

A record type can be in one of two forms. It can be an 8-character string if you use the FACS interface or it can be an equate value if you use the FACE interface. If it is a character string, the name is hashed and an index into a hash table is created. The hash table entry points to a name table entry. If the record type name in the name table does not match the record type name that was specified, a linear search is performed until the correct name table entry is found. The name table entry then points to a split chain header in the split chain header array. If the record type is an equate value (FACE interface), this value is used directly as an index into the array of split chain headers. The hash and name tables are not used by the FACE interface. Once the split chain header is found, the dispense mode indicator is checked to determine which type of FARF address is currently being dispensed. The current dispense mode determines which chain of splits to follow. If the record type is shared, the split chain header points directly to the chain of splits as shown in Figure 2. If the record type is unique, the split chain header points to a matrix that contains pointers to unique sets of splits. If splits do not exist for the current dispense mode, the alternate dispense mode splits are used. The input ordinal number is used to determine which split in the chain of splits maps the input ordinal number. If the requested ordinal is not in the range that is mapped by the current split, a pointer to the next mapping is used to access another split. When the split in which the requested ordinal is mapped has been located, the descriptor that is contained in the split (either a band or UFT/FTI) is concatenated with the result of subtracting the requested ordinal from the first ordinal mapped by this descriptor. (This subtraction gives the relative position of the requested ordinal in the descriptor mapping.) Figure 2 shows this process.

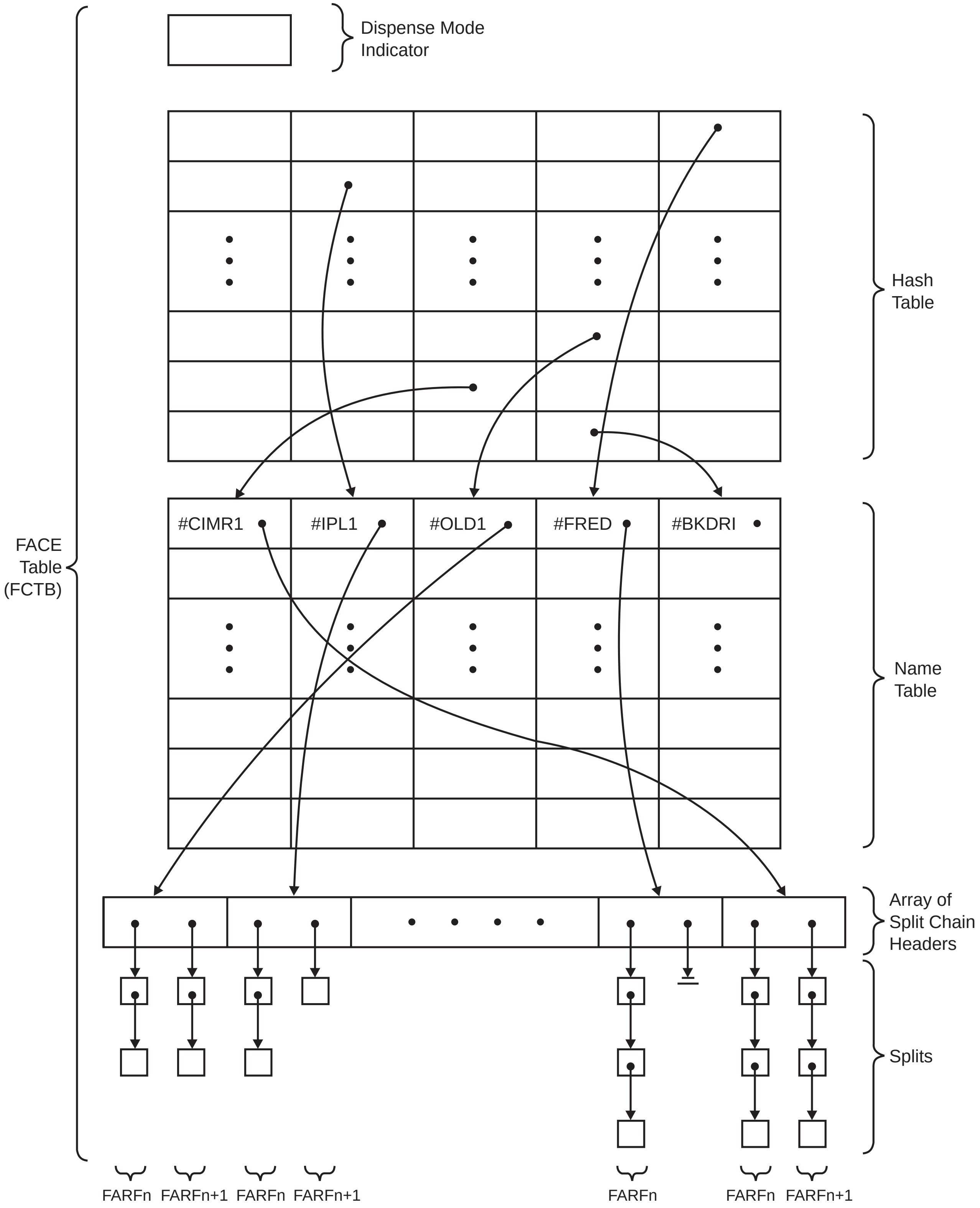


Figure 2. FARF Address Generation

FARF addresses are given to application programs to symbolically refer to specific locations in the database. When a find or file request is made to the database, the FARF address must be converted to a physical location reference. When a FARF address needs to be used to access a physical location, the descriptor field of the address is used to index into a conversion table. A band is a direct index into a band conversion table; the UFT/FTI conversion is a 2-stage lookup. The pointer that is retrieved from the appropriate conversion table addresses the first FACE table split where this descriptor begins mapping the record type. Adding the starting ordinal in this split to the ordinal field in the FARF address re-creates the ordinal number in the record type. If the FACE table split that is accessed first from the pointer in the conversion table does not map the reconstituted ordinal, the next field is used to scan for the correct split. With FARF3 pool addresses, there is no conversion table; the control bits of the address are used to determine the pool type. The PDSCA is used to convert the pool type to pool record type equate value. The rest of the file address is the pool ordinal number. Once the record type and ordinal are known, the process described above to create a file address is used to find the correct split. The base DBON in the correct split is added to the record type ordinal number to produce the actual value that is used to map to a physical location on the database. Figure 3 shows the decoding of FARF3 fixed addresses. Figure 4 shows the decoding of FARF4, FARF5, and FARF6 addresses. See *TPF Concepts and Structures* for a detailed discussion of the database allocation rules.

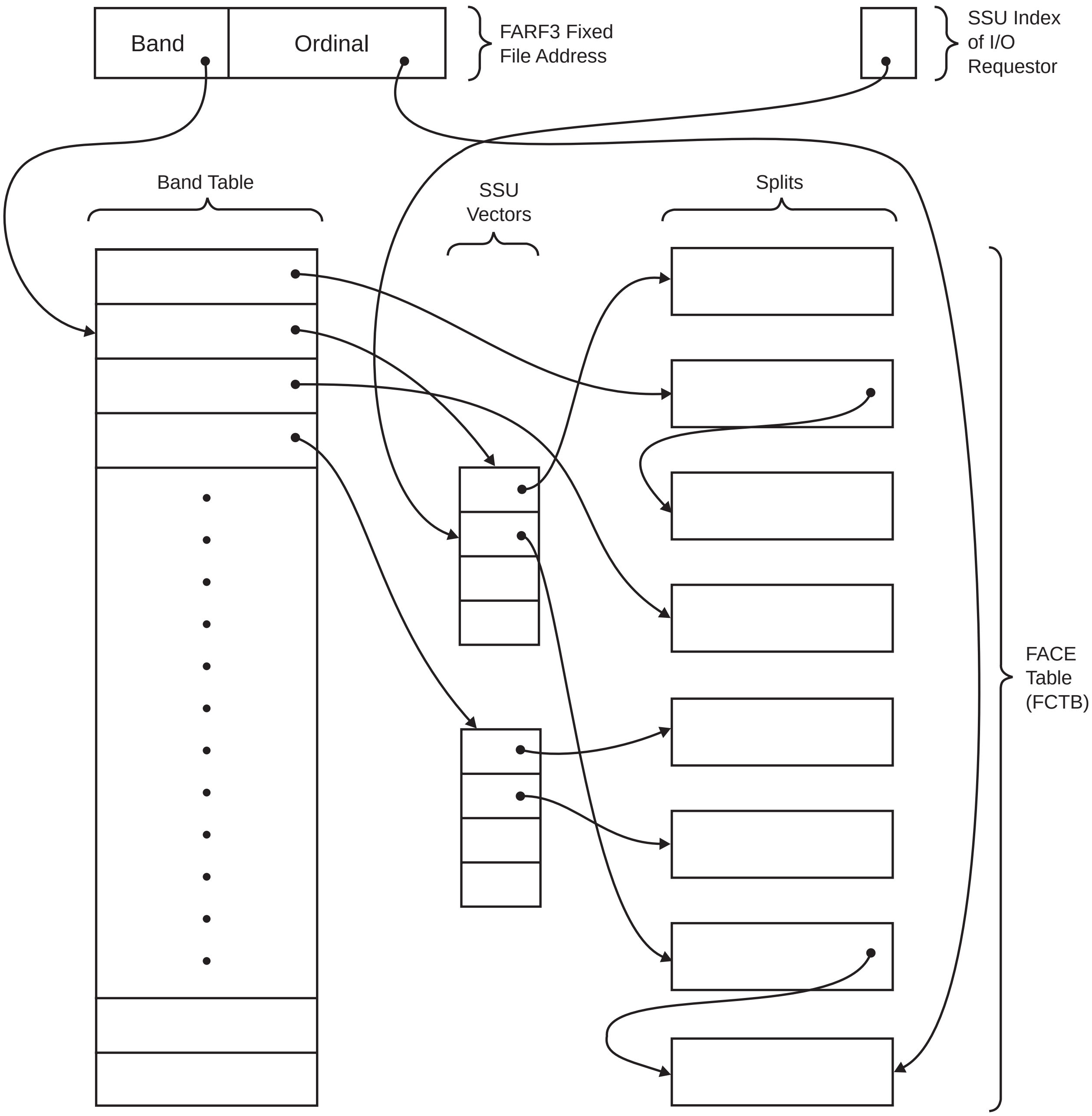


Figure 3. FARF3 Fixed Address Decode Procedure

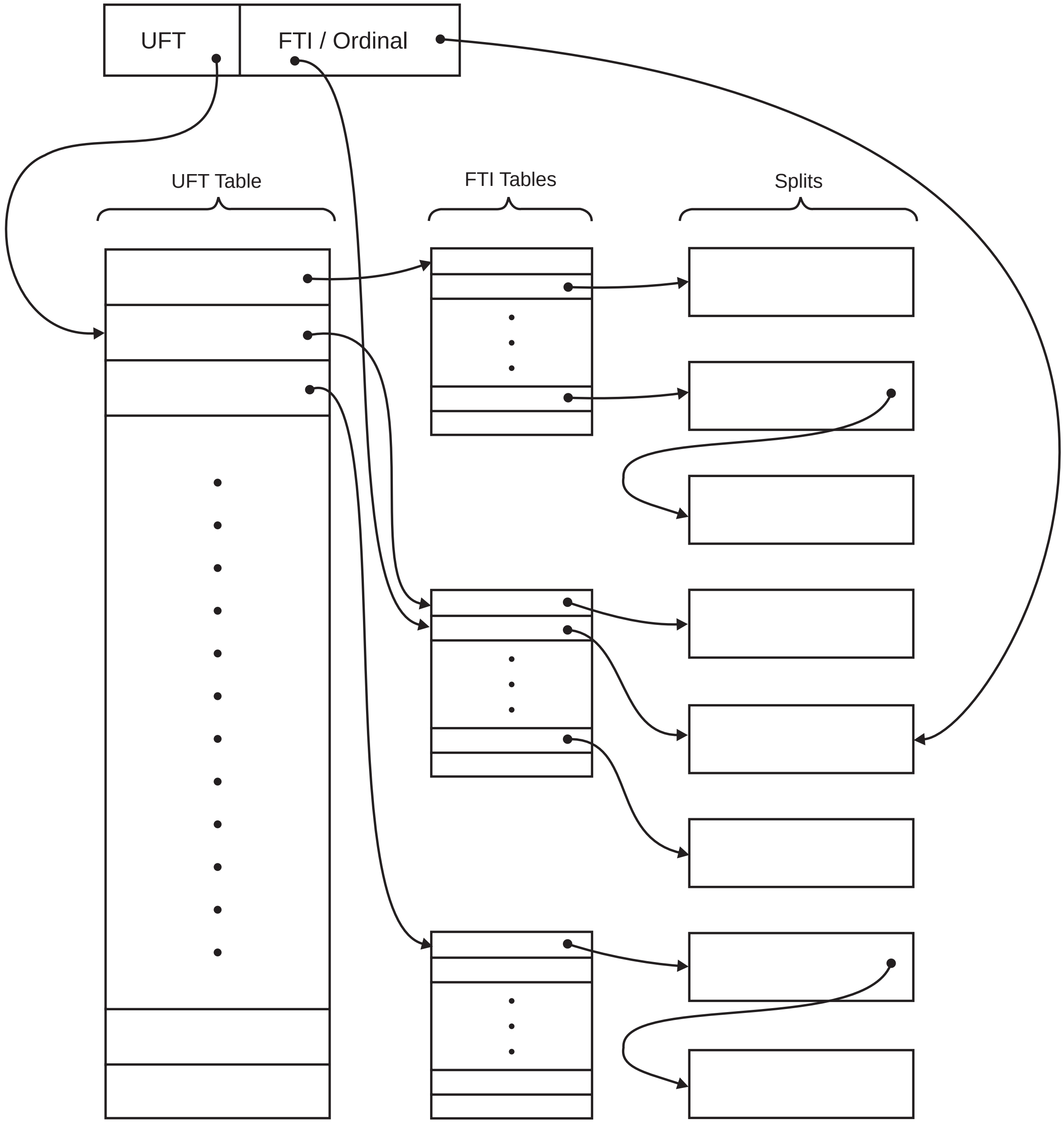


Figure 4. FARF4, FARF5, and FARF6 Fixed Address Decode Procedure

The FARF3 format for fixed files is band number dependent. FARF4, FARF5, and FARF6 formats for both fixed and pool files are UFT/FTI dependent. Band and UFT/FTI numbers must continue to be associated with the same record types from one system generation to another. This ensures that any embedded FARF address will continue to refer to the same record even though the record may have been physically relocated.

The FARF addressing scheme was designed so that migrations would be easier. FARF4 and FARF3 are used together. When all the FARF3 addresses in the system have been converted to FARF4 addresses, the system can be made to run in FARF4 mode only. Similarly, when FARF5 addresses are added to the system, FARF4 and FARF5 addresses can coexist. When all the FARF4 addresses have been converted to FARF5, the system can be made to run in FARF5 mode only.

This makes it easier to convert large databases all at once. FARF6 is independent of FARF3, FARF4, or FARF5, and can be used concurrently.

General File Addresses

A TPF general file is a sequentially organized set of data that, in principle, is similar to a TPF general data set. General files, however, are not MVS-compatible and preceded general data sets in the evolution of the TPF system. There are limited classes of general files used by system programs. For example, the general file that is used to load the pool directories is not compatible with general data sets. However, all general files use a sequential data structure and, in principle, correspond to the general data set procedures.

The general file address format that is supported by the FIND and FILE macros is shown in Figure 1 as General File RRN.

General Data Set Addresses

A TPF general data set is directly related to the meaning of an MVS data set. The records of a data set are allocated sequentially in the same module (or MVS volume). A special macro, file data chain transfer (FDCTC), permits you to process records that are not restricted to the standard TPF file record sizes. A file with standard record sizes is processed online by using the TPF standard find and file type macros.

File referencing for general data sets is slightly different from the procedures used to access fixed record types and pool records. Two general data set macros are provided to allow ECB-controlled programs to access records in a general data set:

GDSNC	Associates a data set name with a unique entry.
GDSRC	Calculates the file address of a specific record in the data set named by the GDSNC macro.

These macros set up a user-specified data level that is appropriate for accessing records in a general data set for the find and file type macros using a physically oriented interface that is transparent to the user.

Extended MCHR File Address

DASDs that are supported by the TPF system require a 7-byte hardware address to physically access a record, plus a 2-byte symbolic module number. This is the address that must be used by channel programs, including those generated for the file data chain macro (FDCTC) and the extended form of the FNSPC and FLSPC macros under the TPF system. Figure 5 on page 12 shows the formats of extended MCHR file addresses used to store and retrieve TPF records.

Hardware File Address

The hardware file address that is required to physically refer to a record residing on a DASD is the same as the extended MCHR file address described previously. The formats are shown in Figure 5 on page 12.

<table>
<tr><th>Bit</th><th>381-Byte Record</th><th>1055-Byte Record</th><th>4096-Byte Record</th></tr>
<tr><td></td><td>Symbolic Module Number;
Must be available at
CE1FMx for the macros.
MM</td><td>Same</td><td>Same</td></tr>
<tr><td>00
15</td><td>2-Byte Bin Number:
Always set to zero.
BB</td><td>Same</td><td>Same</td></tr>
<tr><td>16
31</td><td>2-Byte Cylinder Number:
Zero is the first
cylinder number.
CC</td><td>Same</td><td>Same</td></tr>
<tr><td>32
47</td><td>2-Byte Head Number:
Zero is the first
head number.
HH</td><td>Same</td><td>Same</td></tr>
<tr><td>48
49
50
51
52</td><td rowspan="2">6-Bit Record Number:
The first record number
on a track is 1.
R</td><td>5-Bit Record Number:
The first record number
on a track is 1.
R</td><td rowspan="4">8-Bit Record Number:
The first record number
on a track is 1.
R</td></tr>
<tr><td>53</td><td>0 = Simplex
1 = Duplex
D</td></tr>
<tr><td>54</td><td>0 = Simplex
1 = Duplex
D</td><td>Spare:
Always set to zero.
S</td></tr>
<tr><td>55</td><td>0
L</td><td>1
L</td></tr>
</table>

Figure 5. Extended MCHR File Address Formats

Note: For the IBM 3390, 8-bit record numbers are used for all TPF record sizes as shown for 4096-byte records in the previous figure.

General Data Sets

The general data set (GDS) support allows a TPF application to read and write MVS BSAM or QSAM DASD data sets that are used to pass data between MVS and TPF. General data sets are BSAM or QSAM multivolume fixed-length record data sets with records in TPF sizes (381, 1055, 4095, or 4096 bytes). The GDS support allows the use of nonstandard record sizes up to 4096, but the user must handle the record accessing using either FDCTC or FNSPC/FLSPC macros.

The GDS support does not include the ability to create or expand a data set; all space allocation and formatting must be done on MVS. The GDS support does include the ability to display all mounted data sets by data set name (DSNAME) or data definition name (DDNAME), and the ability to scan the volume table of contents (VTOC) for a specific data set name or to display all data sets with extents on the volume. Also included in the GDS support is the ability to automatically premount a set of data sets and the ability for the system to automatically remount previously mounted data sets during a system restart. It extends the use of the find and file macros to these *general data sets* (referred to as *data sets* in the remainder of this chapter), allowing for TPF single record processing of this data.

Despite the similarity in names, general data sets are not related in any way to general file data sets. General file data sets are designed for use solely by TPF systems for such varied purposes as:
- Schedule change
- Fare/quote ticketing.

General files are discussed in “General Files” on page 17.

The physical attributes of general data sets are as follows:
- File resident on DASD devices
- Data set names (1 to 44 characters)
- Single and multiple volume data sets (16 extents per volume with up to 64 multiple volume sequence numbers)
- TPF record sizes (381, 1055, 4095, or 4096 bytes only)
- TPF record number format
- MVS record number format.

Functions provided by this general data set facility are:
- Premount of data sets from an initial IPL.
- Remount of data sets after restart IPL. This function ensures that all data sets that were mounted before the restart was initiated will again be mounted and available to the system.
- Operator mount and dismount of data sets with commands.
- Single record processing using the find and file macros with the GDS=Y parameter.
- Operator display of mounted data sets with a command.
- Operator display of the VTOC of an MVS volume with a command.

Multivolume data sets can be accessed as follows:
- All volumes of any given data set are accessed as a single contiguous data set. This requires that all volumes of this data set be mounted.

- Single volume processing of any volume in the data set restricts the ECB to the specified volume for a given data level, but allows access to other volumes of the same data set on separate data levels.

Data sets with multiple extents appear as a single contiguous extent on that volume on which the data set resides.

Premount of General Data Sets

An operator message exists to force premount processing to occur on the next IPL of the system. During this IPL the user may want a number of data sets to be mounted. This premount function allows a user to assemble, in the control records provided, the data set names, volume sequence numbers, and volume serial numbers of those data sets. This function is provided to free the computer room operator from manually mounting all the data sets that may be required by the system.

The premount records are contained in program records. The first premount record is CVZE, and the second (if needed) is CVZF. The macro GDSPC is used to build the premount. A GDSPC macro is coded for each data set to be mounted. Each premount record can hold a variable number of data sets depending on the number of volumes per data set.

Once the macros are coded, the program is assembled and loaded to the system. When the premounts are ready for use, the operator issues the ZDSMG INIT command. This causes the GDS restart to process the premount records on the next IPL. Any previously mounted data sets are forgotten.

General Data Set Commands

The general data set support provides the following commands for controlling the system use of general data sets.

ZDSMG MT	Mounts a data set or a volume of a data set
ZDSMG DM	Dismounts a data set or a volume of a data set
ZDSMG DISPLAY	Displays currently mounted data sets
ZDSMG VTOC	Displays the data set names contained on a DASD volume
ZDSMG INIT	Forces reinitialization of the restart records from the premount records on the next IPL
ZDSMG DEF	Defines a data definition name to the system for a tape device, general file data set, virtual reader, or another user-defined media
ZDSMG REL	Removes a data definition from the system
ZDMFS	Shows the currently mounted GDS DASD volumes.

See *TPF Operations* for more information about the general data set commands.

Record Processing of General Data Sets

The following discusses:
- Considerations for the record format of a general data set
- The macros used in processing a general data set.

Record Format

When laying out a GDS record, the record format should take into account the following normal TPF procedures:

- TPF tests the first 2 bytes of the record as a record ID on all file macros. Therefore, the first 2 bytes of the record must be the same as the record ID field in the file address reference word.
- TPF overlays bytes 4-7 with the program name of the program issuing the file macro.
- TPF overlays the last byte of a 4096 byte record with a flag byte.

Processing Macros

A get general data set entry macro (GDSNC) is provided by TPF to allow an application program to retrieve the information necessary to access a specific data set by initializing the appropriate data level or data event control block (DECB) in the ECB. The GDSNC macro must be issued before subsequent find/file macros and returns the file address of the first record.

The parameters provided with the GDSNC macro allow data records to be retrieved in varying sequences within a particular data set. For example, a data set may be accessed by name only regardless of the number of volumes mounted. In addition, any single volume of a data set may be specified for access.

The get general data set record macro (GDSRC) is used to get subsequent record addresses into the ECB data levels or DECBs for use by find and file macro processing. Any number of ECBs may simultaneously address a data set.

Concurrent updating of a data set among loosely coupled TPF images is supported.

See *TPF General Macros* for more information about the GDSNC and GDSRC macros. See *TPF Concepts and Structures* and *TPF Application Programming* for more information about DECBs.

General Files

A TPF general file is a single extent of contiguous space on a disk pack formatted for use within a TPF environment. A general file data set is a subdivision within that general file. The general file data set contains sequentially organized, but randomly accessible, records of related data.

There can be no more than three general file data sets in a general file. General files are subsystem unique, and each general file data set has an installation-specified number, unique within the subsystem, which identifies it to the user and to the TPF system.

Despite the similarity in names, general file data sets are ***not*** related in any way to general data sets. *General data sets* are essentially MVS data sets, and are used primarily to pass data between TPF and MVS systems. General data sets are discussed in "General Data Sets" on page 13.

General file data sets, on the other hand, are designed for use solely by TPF systems for such varied purposes as:
- Schedule change
- Fare/quote ticketing.

From an online application standpoint, general file data sets have certain advantages over general data sets in that:
- They do not have to be opened and closed.
- The instruction path to access them is more direct than that to access general data sets.
- They can be displayed and altered online with TPF commands (for example, ZDFIL, ZAFIL).

Records in a general file data set are not duplicated and, within a general file data set, must be of the same length, either 381, 1055, or 4KB bytes.

The TPF online file formatter must be used to format a general file, and the starting address of each data set must be recorded in the general file module table (GFMT).

A SIP skeleton (SKCVZD) creates program segment CVZD, the general file definition record that also serves as the general file premount record. General files are defined to the system by adding entries to CVZD and then reassembling and loading CVZD to the online system. The GFMT is initialized from CVZD during IPL when CTKB has been loaded with the initial IPL switch (CK9IPLR) set to X'00'.

Positions 42 through 47 of the volume label (cylinder 0, head 0, record 3) are reserved to record the data set numbers that reside on the general file, and positions 48 through 51 record the subsystem ID. The valid range of data set numbers is 00 through 59, but they do not need to be either consecutive or sequentially ordered.

The data set numbers are used to mount or dismount the data sets. Each data set on a general file disk pack has to be mounted and dismounted separately by the operator. The mount function verifies that the data set number is in the volume label and makes the data set available to an online application program; the dismount function cancels that availability. A bypass option is provided to allow a data set to be mounted without checking the volume label for the data set number.

The system generation process does not place the data set numbers or subsystem ID in the volume label of a general file; rather, the information is placed there by a ZAGFL command. Therefore, the user must mount the data sets with the bypass option until a ZAGFL command is issued to place the necessary data in the label.

Application programs may access a mounted data set with any of the find or file macros, except find single (FINSC P/D) and file single (FILSC P/D), by specifying the GDS=Y parameter on the macro. FNSPC and FLSPC macros do not have a GDS parameter since the target module number is not part of the coded file address. A pseudo module number, constructed from the data set number, is used in place of a symbolic module number for I/O. The valid range of pseudo module numbers is defined at system generation, and is accessible to users in the CONKC macro fields @02GDF and @02GDA. Default pseudo module numbering starts at 230, and the range of pseudo module numbers will be 230 through 230+n for data set numbers 00 through n.

The file addressing formats for processing general file records are:
- Extended MCHR
- Relative record number.

Relative record numbers are device independent; programs using this format do not have to be recorded or reassembled to change a general file from one device to another. Relative record numbering is sequential and starts with 0. Whichever address format the user chooses, records can be read or written either sequentially or randomly by record.

The RAISA macro is provided to increment file addresses in the MCHR and relative record formats by a user-specified count. It may be invoked by either online or offline segments. The generated code accounts for control bits and track overflow in performing the addition, and the result is returned to the caller in the specified data level. When RAISA is called from an offline program with a relative record number as input, a pointer to an 8-byte address suitable for Seek and Search I/O commands is also returned.

The status of a general file data set remains the same over a restart. If it is active at cycle-down time, it will be ready for use again when restart completes.

The following commands allow the operator to work with general file data sets and labels:

ZFMNT	Mount a general file data set
ZFDNT	Dismount a general file data set
ZDMFS	Show mounted general files
ZDGFL	Display a general file label
ZAGFL	Alter a general file label.

See *TPF Operations* for more information on these commands.

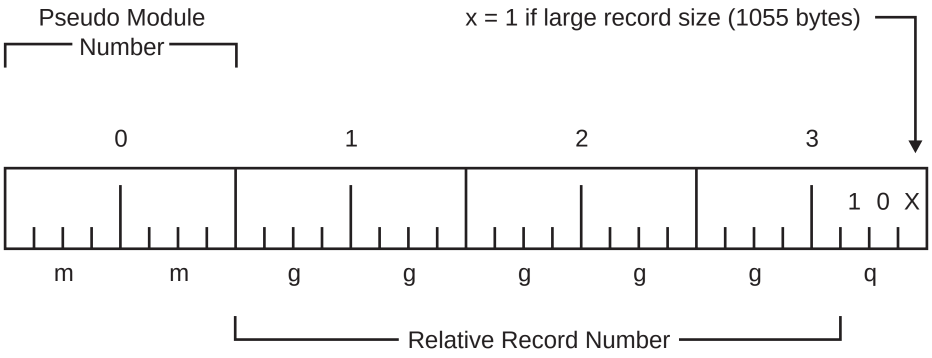
Programming Notes for General Files

The following describes the file addressing formats that can be used to read a general file data set (using FIND or FIND SPECIAL).

Note: In these descriptions, the pseudo module number is defined as the sum of the starting general file module number and the number of the data set the user wants to access. For example, if the TPF system is generated to start general file pseudo module numbers at 010 (see the GFMOD parameter of the RAM macro in *TPF System Generation*), then general file data set number 03 will be pseudo module number 013. The module number that the user specifies will be the hexadecimal equivalent of 013, that is, 0D.

Processing a Relative Record Number Request

The parameter *mmgggggg* that must be provided for a general file relative record number request has the following format:



If the calling segment is processing a relative record request (as in the ZDFIL *mmggggggq* format):

- *mmggggggq* must be placed in fields CE1FMn, CE1FCn, CE1FHn, and CE1FRn of the appropriate entry control block (ECB) data level fields or IDECFM, IDECFC, IDECFH, and IDECFR of the appropriate data event control block (DECB).
- The caller must issue a FIND using the GDS=Y keyword.

Processing an Extended MCHR Request

If the calling segment is processing a request for an extended MCHR address (as in the ZDFIL *mmmmccccchhhhrr* format):

- CE1FXn bytes 0-1 must contain hex zeros.
- CE1FXn bytes 2-3 must specify the cylinder address.
- CE1FXn bytes 4-5 must specify the head address.
- CE1FXn byte 6 must contain the record number.
- CE1FMn bytes 0-1 must contain the pseudo mod number.
- The caller must issue an FNSPC macro specifying extended addressing (E parameter).

File Pool Support

Random file pool storage support (referred to as file pool support in the remainder of this chapter) enables a program to obtain temporary file storage space in much the same way as temporary main storage is obtained. File addresses for this temporary file storage are obtained by application programs using a macro. These programs can then store data at the disk locations for whatever period of time is appropriate. Another application macro indicates that a file pool address can no longer be considered in use and is available again.

File pool storage supports devices that use file address reference format (FARF) addressing (for example, the 3380 DASD).

This chapter is an overview of file pool support with descriptions of the following:
- Functional description
- File address formats
- Pool characteristics
- Maintenance functions
- Management functions.

Functional Description

It is not unreasonable to envision systems with millions of temporary data records maintained in file pool storage areas. Consequently, an effective and dynamic facility is required to maintain and manage a file pool complex capable of satisfying varying requirements in the exacting environment of a real-time system.

All pool addresses are assigned unique bit status indicators in an array. By interrogating a specific bit, a program can determine if a file record is available for use or if it is being used for the storage of data. Because of the enormous requirement for temporary data storage records, a complete array is prohibitive for an online system in terms of main storage utilization. Therefore, an effective method is necessary to maintain and manage a bit array that represents a large number of records.

The medium used by the pool facility for containing a bit array is called a *directory record*. A pool bit array is generally first constructed from information in the FACE table (FCTB), which is created by the FACE table generator (FCTBG). The bit array is then broken down into directories, each having a record header. The information in the record header provides a means of locating the bits in each directory in the full bit array.

To improve overall pool performance, the TPF system allows multiple directories to be brought into memory in sets called *directory sets*. These sets are kept in two directory set buffers called the *active* and *standby* directory set buffers.

Directory set buffers are carved (or allocated in main storage) at IPL time by the initializer program (CCCTIN). These buffers will be carved in high storage on 4–KB boundaries. The number of directories in each set defaults to 1. The number of directories per set can be changed by the user with the ZGFSP command. See *TPF Operations* for more information about the ZGFSP command. Based on feedback from a data reduction report, users determine an optimum assignment for the number of directories in a set, which is a trade-off between working storage usage and get/release file pool address activity. For additional information about determining the minimum number of pool directories, see *TPF System Generation*.

During cycle-up, pool restart code fills both of the pool buffer areas for the long-term sections, but fills only the active buffer for the short-term sections.

The following values are associated with directory set size:

Value	Description
ACTIVE	The number of directories in the set currently dispensing addresses.
STANDBY	The number of directories in the set about to dispense addresses.
NEW	The size to which the active and standby set sizes will change through reorder processing.
CARVE	The size, in directory increments, that is available to hold directories. This value represents the amount of storage carved by CCCTIN for the directory buffer. Set size cannot exceed this value until the processor is re-IPLed, allowing CCCTIN to allocate more space.

At IPL time, CCCTIN looks at the 3 set size values in keypoint 9 (CTK9):
- ACTIVE
- STANDBY
- NEW.

Using the largest of the three set size values, CCCTIN carves memory. This area contains the pool directory set control area (ICY7PR) and, if carving for a short-term pool, will also contain:
- Short-term limits control subtable (ICY8CS)
- Short-term processor control records (ICY$PR) work area
- Short-term master control file (CY$CR) work area.

File Address Formats

The file address formats supported for pool record addressing are described in “File Address Formats” on page 3.

Pool Characteristics

File pool support is provided for external storage devices that use FARF addressing. Support is included for small/large/4K, single/duplicate records with various retention requirements. There are 10basic pool types, as follows:
- Small, long-term records (SLT)
- Small, short-term records (SST)
- Small, long-term duplicate records (SDP)
- Large, long-term records (LLT)
- Large, short-term records (LST)
- Large, long-term duplicate records (LDP)
- 4K long-term records (4LT)
- 4K short-term records (4ST)
- 4K long-term duplicate records (4DP).
- 4K long-term duplicate FARF6 records (4D6).

Small, large, and 4K records are defined as 381, 1055, and 4095 bytes long respectively. Pool types SLT, SST, LLT, LST, 4LT, and 4ST contain single copy records in a partially duplicated file system and duplicate records in a completely duplicated file system. Pool types SDP, LDP, 4DP, and 4D6 are provided for systems with a partially duplicated file system and a requirement for duplicated file storage records.

Long-term file storage records can be maintained during an interval of time as determined by operational procedures. Short-term pools, however, are designed for quick turnover records. Maintain short-term file storage records for short intervals of time (in other words, the time required to complete a transaction with a customer). When returned by application programs, long-term addresses are batched to an online data file to be returned to the TPF system at a later time by recoup or pool directory update (PDU) processing. Short-term addresses are, however, immediately returned to the relevant pool for reuse.

Each basic pool type applies to a specific device and uses FARF addressing. Each group of pool types on a specific device is called a *pool section*. Each short- and long-term pool can have a maximum of four pool sections. For example, the following represents four pool sections for small, long-term records:

- SLTA
- SLTB
- SLTC
- SLTD

where A, B, C, and D represent the device type defined with the system initialization program (SIP). See *TPF System Generation* for information about SIP.

Pool sections can be broken up into noncontiguous areas on a device type. Each of these areas is called a *pool segment*.

For example, the small, short-term pool (SST*x*) might consist of 2 pool sections, each with segments as follows:

- 3390 SSTA pool section with 2 segments
- 3380 SSTB pool section with 2 segments.

Each directory set is initialized when needed and updated when short-term pool addresses are returned to those pool sections. Because short-term directories are recycled, they are not included in recoup and pool directory update (PDU) processing. When a short-term pool section is depleted, the relevant directory sets are again reinitialized and used one-at-a-time starting with the pool section’s first directory. This is called *recycling the pool section*.

Maintenance Functions

This section provides a general description for the following pool maintenance functions:

- Pool generation and reallocation
- Pool directory update (PDU)
- Recoup support
- File pool count reconciliation
- Online directory capture and restore

Pool Generation and Reallocation

Input to pool generation is in the form of constants and equates in the system configuration macro (SYCON) built by the system initialization program (SIP). See *TPF System Generation* for details about SIP. Pool support allows as many as 4 device types that can be used for pools. SYCON values should be specified for all equates to prevent assembly errors even though fewer than 4 devices exist on the system.

The labels pertinent to pool support are shown in Table 1.

Table 1. SYCON Values for Pool Support

Label	Description
&CGDEV(1) &CGDEV(2) &CGDEV(3) &CGDEV(4)	These are global SET symbols used to assign each device a name. They must be set to blanks if the device is not supported. Otherwise, they must contain the device name. For example, if 3350s are supported as the first device for pools then &CGDEV(1) = 3350.
CSONCMA CSONCMB CSONCMC CSONCMD	These define the number of cylinders per module for pool devices 1, 2, 3, and 4 respectively.
CSONHCA CSONHCB CSONHCC CSONHCD	These symbols define the number of tracks per cylinder for each device.
CSONSRA CSONSRB CSONSRC CSONSRD	These symbols define the number of small records per track for each device.
CSONLRA CSONLRB CSONLRC CSONLRD	These symbols define the number of large records per track for each device.
CSON4RA CSON4RB CSON4RC CSON4RD	These symbols define the number of 4KB records per track for each device.
CSONPNMA CSONPNMB CSONPNMC CSONPNMD	The number of modules used for pool records in each device type. These values are used with the increments (CSONPLBA, CSONPSBA, and so on) to determine the small, large, and 4KB file address ranges for each device type. These ranges must not overlap among device types; otherwise, an ambiguity could be created when trying to associate an address with a device type.

Pool definition information is part of the FACE table that is generated by the FACE table generator (FCTBG).

The record code check (RCC) is used throughout pool support to identify specific pool types and to index pool-specific tables in the control program. The RCC indicates the pool type as shown in Table 2. The intersection of pool type and device type gives the RCC. Support is provided for 4 device types.

Table 2. Record Code Check Values

	Device Type			
Pool Type	**A**	**B**	**C**	**D**
Small Long-Term	04	28	4C	70
Small Short-Term	08	2C	50	74
Small, Long-Term Duplicate	0C	30	54	78

Table 2. Record Code Check Values (continued)

Pool Type	Device Type			
	A	B	C	D
Large Long-Term	10	34	58	7C
Large Short-Term	14	38	5C	80
Large, Long-Term Duplicate	18	3C	60	84
4K Long-Term	1C	40	64	88
4K Short Term	20	44	68	8C
4K Long-Term Duplicate	24	48	6C	90
4K Long-Term Duplicate FARF6	94	98	9C	A0

The beginning ordinal number is the ordinal number assigned to the first record in the pool segment. Successive ordinal numbers are assigned to the remaining records in the segment. These ordinal numbers must be unique in the pool type. In addition, in FARF3/FARF4 mode, because large and 4-KB records share ordinal number ranges, there can be no ordinal number duplication in these two pool types when they also have equivalent retention and duplication definitions.

Pool Generation and Reallocation Procedure

The pool generation and reallocation procedure can be used to create an initial pool configuration or change an existing pool configuration. If you are deleting pool files from your configuration, you must deactivate pool directories before changing your existing pool configuration.

Note: The pool generation and reallocation procedure can only be used if all processors in the complex are running 32-way loosely coupled pool support and the pool data structures have been converted to 32-way loosely coupled pool support format.

See *TPF Migration Guide: Program Update Tapes* for more information about migrating to 32-way loosely coupled pool support and converting pool data structures to 32-way loosely coupled pool support format.

To generate or reallocate a pool configuration, do the following:

1. Enter the ZPMIG command with the STATUS parameter specified to verify that all processors are running 32-way loosely coupled pool support and that the pool data structures are in 32-way loosely coupled pool support format.

   See *TPF Operations* for more information about the ZPMIG command.
2. Define the online database using RAMFIL statements that are available in the system initialization program (SIP). If you are deactivating pool directories, specify the DEACTIVATE parameter with the RAMFIL macro to allow pool segments to be deactivated.
3. Run the FACE table (FCTB) generator to create generation input.
4. Enter **ZPOOL GENERATION CREATE** to create initial pool directories or to change existing pool directories.
5. Make sure all processors are in 1052 state.

8-Byte File Address Support

To activate FARF6 8-byte file addressing, enter **ZMODE 6** so that 4D6 pools can be generated. See *TPF Operations* for more information about the ZMODE command.

6. Enter **ZPOOL GENERATION RECONFIGURE** to reconfigure existing pool directories.

   Note: If you do not enter ZPOOL GENERATION RECONFIGURE, all existing pool files will be made available. Enter **ZPOOL GENERATION RECONFIGURE** unless you are creating your initial pool directories.
7. Load the new FCTB that was created by the FACE table generator.
8. Enter **ZPOOL GENERATION UPDATE** to update the pool directories.
9. When prompted by the TPF system, do one of the following:
   - Enter **ZPOOL GENERATION ONLINE CONTINUE** to confirm and continue rolling new or changed pool directories into the pool rollin directory (#SONRI).

     Note: Entering ZPOOL GENERATION ONLINE CONTINUE causes a CTL-3C system error dump to occur, which requires a TPF system IPL.
   - Enter **ZPOOL GENERATION ONLINE ABORT** to stop new or changed pool directories from being rolled into the pool rollin directory (#SONRI).
10. Make sure the TPF system is in 1052 state. If necessary, cycle the TPF system to 1052 state.
11. Enter the following commands to verify that the new or changed pool directories were rolled into the pool rollin directory (#SONRI) as you wanted them rolled in.
    - **ZPOOL DISPLAY**
    - **ZDFPC**
12. Do one of the following:
    - If the pool directories were rolled in correctly, do the following:
      a. Cycle the TPF system to NORM state.
      b. Enter **ZPOOL INIT PSDIR** to initialize the pseudo directory records before the next recoup run. Once you enter ZPOOL INIT PSDIR, you can no longer restore pool rollin directory (#SONRI) records by entering ZPOOL GENERATION FALLBACK.
    - If the pool directories were not rolled in correctly, do the following:
      a. Enter **ZPOOL GENERATION FALLBACK** to copy the recoup SONRI save area (#SONSV) back to the pool rollin directory (#SONRI).
      b. When prompted by the TPF system, do one of the following:
         - Enter **ZPOOL GENERATION ONLINE CONTINUE** to confirm and continue rolling fallback directories into the pool rollin directory (#SONRI).

           Note: Entering ZPOOL GENERATION ONLINE CONTINUE causes a CTL-3C system error dump to occur, which requires you to IPL all processors in TPF system complex.
         - Enter **ZPOOL GENERATION ONLINE ABORT** to stop the fallback directories from being rolled into the pool rollin directory (#SONRI).

13. If you are deactivating pool directories, do the following:
    a. Enter the ZSDEA command.
    b. When the pool directories you want to delete are no longer in use (deactivated), run this procedure again to define the changed online database.

Pool Directory Update (PDU)

Pool directory update (PDU) functions are run by doing the procedures that follow.

PDU Procedures

1. Enter the ZRPDU CREATE command to start PDU processing.
2. Once the ZRPDU CREATE command has completed processing and offline multiple releases have been identified, enter the ZDUPD command, specifying the S parameter to verify the PDU pseudo directory (#SONUP) against the pool rollin directory (#SONRI).
3. Once the ZDUPD command has completed verification, enter the ZDUPD command, specifying the C parameter to continue with the rollin process.

Recoup Support

The recoup function is used to reconcile long-term file pool directory records with the actual status of the file pool records. See “Recoup” on page 103 for more information about recoup.

File Pool Count Reconciliation

File pool count reconciliation functions are run by entering the ZRFPC command.

Online Directory Capture and Restore

Online directory capture and restore functions are a part of normal recoup processing and can also be run by entering the ZRDIR CAPTURE and ZRDIR START RESTORE commands.

Management Functions

File pool management functions are supported by the following facilities designed to enhance file pool processing:
- Get file storage (GFS)
- Release file storage (RFS)
- Initialize file pools
- Pool function switches
- Pool commands

These facilities are described in the following sections.

Get File Storage (GFS)

Application programs issue the get file storage macros to obtain large or small file pool records respectively. The following sections describe the functions available in the processing of these macros.

Basic Pool Selection

Selection of a pool section is normally based on the get file storage macro expansions. Pool fallback and ratio dispensing fallback can modify normal pool selection as described in the following sections. See *TPF General Macros* for more information about the file storage macros.

Pool Fallback

When a depleted pool section is selected for address dispensing, an alternate compatible pool section is used if possible. The appropriate GFS program is started for processing the alternate pool section.

Selection of alternate pool sections for fallback processing is done based on either a specified or predefined schedule. Systems with pools have predefined schedules for short-term pool section fallback. These predefined schedules refer to duplicate pool sections only. Therefore, a system with pool sections has no predefined short-term to short-term pool fallback. However, optional, primary pool fallback schedules can be defined for systems with pool sections. If specified, these optional schedules can provide short-term to short-term, short-term to long-term, or long-term to long-term fallback capability for depleted pool sections.

The optional fallback can be specified in the SYCON macro or by a ZGFSP command. See *TPF Operations* for more information about the ZGFSP command.

Ratio Dispensing

This facility is used to dispense addresses for any basic pool type using several sections of the pool (for example, 3390 SSTA pool sections). When a pool section is selected, its ratio factor (that is, number of addresses) is used to determine how many addresses to dispense from that section before selecting another pool section from the schedule. Therefore, ratio dispensing allows any specific record ID (pool record) to be spread across several device types.

Bit Scanning

The GFS programs process each bit in a directory as an entity although groups of bits are selected for ease of processing.

Dispensing Algorithms

Dispensing of file pool addresses is based on the pattern of available bits (records) in the directory records. With all bits in available status, the following describes the GFS algorithm.

Pool addresses are dispensed across all pertinent modules by increasing the ordinal by 1 because each ordinal is a continuous number.

Directory Replenishing

Multiple directory records are generated for each pool section. Directory replenishing (or reordering) for short-term pool sections generally consists of scheduling retrieval of a new directory set before the in-use directory set is depleted of available addresses. This is done when the remaining number of available addresses in the in-use directory set reaches a predefined critical level called the *reorder level*. At this point, a control transfer is used to schedule an ECB and the retrieval of a directory reorder program to perform this function.

The short-term pool sections are designed to be recycled. Therefore, when the last directory set of such a pool section is depleted, the first directory of the pool section set is again set up for dispensing pool records with all bits in available status.

For long-term pool sections, two sets are maintained in main storage. These sets are referred to as the *active* and the *standby* sets. When the active set is depleted, the standby set becomes active and a control transfer is used to schedule a reorder of the depleted set.

Implied Wait Processing

When a depleted directory is referred to in order to satisfy a get file storage macro request, the requesting program is placed in a CPU Loop queue to delay processing of the request. The get file storage macro is processed again when the program is again given control by the CPU Loop program.

This implied wait procedure is linked to pool sections that require directory reorder support. The procedure is enforced to allow retrieval of a new directory to replace an empty in-use directory.

GFS Keypointing

Each pool section is assigned a counter to determine when a file update of critical keypoint information should be done. After an address is dispensed from a pool section, the keypoint update counter of that pool section is checked. If a keypoint update is necessary, a keypoint procedure is started. The respective keypoint update counters are also reset.

Module Down Processing

GFS does not dispense pool addresses referring to disk modules in down or offline status. One of the following techniques is used to recognize the condition and allow dispensing to continue until a pool address is found that can be dispensed:

- Long-term addresses are marked in the directory as being dispensed and the address of the directory is passed to return processing.
- Short-term addresses are set to unavailable status and skipped.

Release File Storage (RFS)

Application programs issue the release file storage macro to release file pool records. The processing done by RFS is based on whether the record is long-term or short-term.

Processing Long-Term Released File Pool Addresses

Long-term releases are blocked into CYSRB tape records. These tape records are written to the real-time tape (RTA).

Processing Short-Term Released File Pool Addresses

Released short-term addresses are returned directly to the respective directory if it is part of a set in main storage and being used by GFS for address dispensing.

RFS Keypointing

RFS performs keypointing procedures similarly to GFS for released short-term pool addresses only. Each short-term address that is returned to its respective directory causes the relevant keypoint update counter to be checked as described previously for GFS keypointing.

Initialize File Pools

Initialization for pool management is done during CP initialization, restart processing, and cycle-up. During CP initialization, the main storage ICY7PRs are carved out of the permanent main storage area for each pool section existing in the system. Pointers to these areas are initialized. Following restart processing, the system is in 1052 state with pool management still inactive though partially initialized. Pool management is inactive in 1052 and UTIL state. During cycle-up to CRAS state or higher, the pool management facilities are initialized. At this point in time, all pool management facilities are available for use by application programs.

Shutdown of File Pools

All pool management facilities are shut down (that is, inactivated) during and as the last step of cycle-down. The file copies of each in-use directory are updated, the last CYSRB records are written to tape, and all keypoints are updated to reflect a planned shutdown of the pool complex. A system re-IPL or cycle-up must follow a cycle-down to again reactivate pool management.

Pool Function Switches

A switch is assigned to each major pool function to indicate if a function is active. This is important because:

- Some functions must not be run concurrently; they are mutually exclusive functions.
- When any pool function is active, a cycle-up or cycle-down must normally be disallowed.

The file pool maintenance and initialization scheduler sets all but the recoup switch, which is set by recoup as needed. The PFSWC macro is used by the individual functions (that is, programs supporting the function) to reset a switch. This macro also defines the switches that are located in field CY5PFS of the pool management global table (CY5GT). You can use a CINFC CMPFSW macro to obtain the address of the switches.

Pool Management Commands

Some pool management functions are controlled with commands. All file pool commands are transferred to the file pool maintenance and initialization scheduler for initial processing. When started in 1052 or UTIL state, this scheduler will normally force a pool management cycle-up and cycle-down sequence to perform initialization generally required for these functions even when pool management is not active. Bypass options are provided to avoid this special initialization procedure for reconciling pool counts and, optionally, for recoup functions.

The following list shows the pool management commands with a brief description of each. See *TPF Operations* for more information about these commands.

ZDFPC
Allows you to display the count of all available file pool addresses. You can display the counts for a specified device or pool type. In addition, you can display pool counts on a time-initiated basis.

ZGFSP
Allows you to do the following miscellaneous file pool functions:

- Modify or display file pool control parameters
- Modify file pool fallback schedules
- Modify the ratio dispensing schedule
- Start or stop get file storage (GFS) functions.

The monitoring function is a GFS function that allows you to monitor long-term pool activities. Start monitoring by entering ZGFSP OPT with the MON parameter.

The pool monitor starts in all NORM state processors in a loosely coupled environment. If the monitor is active when a processor cycles to NORM state, the function automatically starts in this processor. The monitor in each processor analyzes pool usage for its processor only. At 1-minute intervals, the pool monitor checks the pool usage for each long-term pool section in the subsystem and displays the appropriate message as follows:

- Message CYGM0006W is displayed if the available address count increases.
- Message CYGM0011W is displayed if the number of pool addresses dispensed in the last minute is greater than an established amount.
- Message CYGM0007W is displayed if, in any three consecutive minutes, the number of addresses dispensed per minute is greater than an established minimum.
- Message CYGM0008W is displayed if the number of available addresses drops below the established minimum for that pool type.

| See *Messages (System Error and Offline)* and *Messages (Online)* for more information.

ZGAFA Allows you to get a file pool address.

ZGAFI Allows you to get a file pool address by record ID.

ZPMIG Allows you to do the following 32-way loosely coupled pool support functions:

- Convert the pool data structures from pool expansion (PXP) support format to 32-way loosely coupled pool support format
- Return the pool data structures from 32-way loosely coupled pool support format to PXP support format
- Display the 32-way loosely coupled pool support migration status of each processor in the complex.

You can enter this command only from a processor where 32-way loosely coupled pool support is installed and active, and only when the processor is in 1052 state or higher.

See *TPF Migration Guide: Program Update Tapes* for more information about migrating to 32-way loosely coupled pool support and converting pool data structures to 32-way loosely coupled pool support format.

Part 2. Caching Support

This part contains information about caching support for the TPF system, including:
- Virtual file access (VFA)
- Record caching.

Virtual File Access (VFA)

Virtual file access (VFA) provides an intermediate staging area between the application program data and the direct access file database. Because the VFA area is in real main storage, much higher access rates can be achieved and input/output (I/O) channel load reduced.

VFA is intended to effectively use the real main storage that cannot be efficiently used by TPF working storage and main storage resident application programs. VFA is also used by the TPF system as an intermediate file storage area to avoid having to maintain information in an entry control block (ECB) about when a file completed. When an application issues a file, the data record is copied into VFA whether or not it is a VFA candidate, unless it is a 4 KB non-candidate record. If the record is not a candidate then the record is filed from VFA by the control program and removed from VFA. If the record is a VFA candidate, then the record is handled according to the rules described below for VFA candidates.

VFA is conceptually similar to the virtual storage page pools used in IBM virtual storage operating systems. Since the number of records competing for residency in VFA storage greatly exceeds the amount of VFA storage available, only the most active records will be readily accessible through VFA. However, the design does allow for some selectivity in the number and type of records that can become resident in VFA so all of the database does not have to compete for VFA storage.

To further relieve I/O channel load, VFA permits multiple updates to be applied to a record resident in VFA without the updates being reflected to the database copy until necessary.

VFA Candidate Records

You can define record groups for VFA candidates with the following attributes:

Delay Filing

When a FILE macro is issued, the record is copied to VFA and not written to file until:

- A cycle down or catastrophic software IPL occurs.
- The VFA buffer has not been referenced and is required for another VFA candidate record (age out).

In a loosely coupled environment, this record is not synchronized between processors. That is, other processors accessing this record will not see the version of this record in any other processors VFA.

Synchronized Delay Filing

When a FILE macro is issued, the record is only physically written to DASD when:

- A cycle-down or catastrophic software IPL occurs
- The VFA buffer has not been referenced and is required for another record (age out).
- Another processor signals that it wants the record.

Immediate Filing

When a FILE macro is issued, the record is copied to VFA and is immediately filed. In a loosely coupled environment, this record is not synchronized between processors. That is, other processors accessing this record will not see the version of this record in any other processors VFA.

Synchronized Immediate Filing
When a FILE macro is issued, the record is copied to VFA and is immediately filed. When another processor signals that it wants to use the record, the copy in VFA is invalidated.

Data record processing occurs for the following general macros:
- FINDC
- FINWC
- FINHC
- FIWHC
- FILEC
- FILUC
- FILNC
- GDSRC.

See *TPF General Macros* for more information about these macros.

The following general and system macros cause the target record to be flushed from VFA if the target record is in VFA:
- FINSC
- FILSC
- FNSPC
- FLSPC.

Note: For records defined as VFA synchronization candidates, flushing is only performed on the processor that issues the FNSPC or FLSPC macro. Flushing is ***not*** performed on all processors. This could lead to database integrity problems.

See *TPF General Macros* for more information about the FINSC and FILSC macros. See *TPF System Macros* for more information about the FNSPC and FLSPC macros.

VFA program record processing occurs for the following general macros:
- ENTRC
- ENTNC
- ENTDC
- GETPC.

See *TPF General Macros* for more information about these macros.

Whenever VFA delay filing is selected and active (both normal delay filing and synchronized delay filing), it is possible for an application program to retrieve a record from file using the FDCTC macro while a more recent copy of the record exists in a VFA buffer. For this reason, it is essential that any application program that manages data records using the FDCTC macro not be processed while delay filing or synchronized delay filing is active.

Note: Delay filing is active only in NORM state when one or more VFA candidate records has been defined with the delay file or synchronized delay filing attribute. You can use the capture and restore utility while VFA is active.

Data records are defined as VFA candidates by record ID. Program records are defined as VFA candidates by record ID as well. Special record ID X'00FF' in the record ID attribute table (RIAT) describes the candidacy of all file resident programs.

Note: Program records are VFA candidates when they are retrieved for the ENTRC, ENTNC, ENTDC, or GETPC macros only.

If a program record that is resident in VFA is accessed by a FILSC, FILEC, FILUC, or FILNC macro, the record is flushed from the VFA buffers.

See *TPF Operations* for more information about the ZRTDM DISPLAY and ZRTDM MODIFY commands, and for information about displaying and modifying VFA candidacy in the RIAT.

VFA Resource Definition

VFA requires the following information about resources:

- The ratios of large (1055), small (381), and 4 KB (4095) buffers and the percentage of each size that is to be maintained on reserve chains.
- The value specified by the user for the buffer reuse threshold value. The counter value is used to determine when a buffer is to be moved from the aging chain to the reserve chain.

VFA main storage occupies the area following the end of working storage, and includes the VFA tables and VFA buffers. The VFA tables are used by the VFA control program to manage its buffers. The VFA buffers are the main storage in which VFA candidate records reside.

There are three types of VFA buffers:

- Small (381)
- Large (1055)
- 4 KB (4095).

The number of buffers allocated for each type is computed by VFA from user-defined ratios. The buffers are chained together in either one of two chains:

- The aging chain
- The reserve chain.

The first chain is the *aging chain*. This chain contains the majority of the records in VFA. When a record is added to VFA, it is copied into a buffer that is allocated from the *reserve chain*, which is the second chain, and then placed on the aging chain. VFA determines the most active records by maintaining a reference counter for each resident record. This counter is incremented each time the record is referenced by a macro. When a record is copied into VFA, the counter is set to zero and the record buffer is placed on the bottom of the aging chain. Each time the record is referenced, the counter is incremented. When the size of the reserve chain falls below the specified percentage, the top buffer on the aging chain is accessed and its reference count is compared to the user-defined value. If the reference counter is greater than the user-defined value, the count is zeroed and the entry is placed on the bottom of the aging chain. Otherwise, the buffer is moved to the reserve chain and if there is a delay file pending in the buffer, the write is scheduled. This checking of the aging chain continues until the reserve chain is at the percentage specified.

Therefore, the aging chain is used to hold the current VFA resident records that may have delay files pending. The reserve chain is used to hold those records that can be overlaid with a new VFA candidate because they were not referenced a minimum number of times and the records do not have a delay file pending. Any record on the reserve chain can still be referenced by an application find.

Note: A commit scope buffer, which is a type of VFA buffer, is not aged out of VFA. The size of the reserve chain should be such that when a delay file buffer is moved from the aging chain to the reserve chain and the write is scheduled, the write is completed by the time an attempt is made to allocate the buffer for a new record. Buffers on the reserve chain are used for both candidates and non-candidate write requests. Buffers used for non-candidate write requests are placed on the top of the reserve chain once the write is completed.

VFA Record Selection

Use the ZRTDM MODIFY command to define fixed file records or pool records as VFA candidates. See *TPF Operations* for more information about the ZRTDM MODIFY command.

You can define each record group with the following attributes:

Delay Filing

When an application program issues a FILEC or FILNC macro for a delay filing candidate, the update is noted in the VFA buffer but the updated record is not physically filed to DASD. This reduces the amount of physical I/O activity when VFA candidate records are repeatedly filed.

The record is physically filed to DASD during the following conditions:

- The VFA buffer is being moved to the reserve chain because it failed the buffer reuse threshold value test and the size of the reserve chain has fallen below the specified percentage. This is known as *force filing*.
- The TPF system is cycling from NORM state after entering the ZCYCL command.
- The TPF system is cycling from NORM state after entering the ZRIPL command.
- The TPF system is cycled to 1052 state after a ZRIPL command with the BP parameter specified is entered or because of a catastrophic system error resulting from a software IPL.

When a FILUC macro is processed, the delay filing attribute is handled in one of the following ways:

- If the TPF system is running on a uniprocessor, delay filing is in effect.
- If the TPF system is running in a loosely coupled environment and the ZRTDM MODIFY command with the LOCKF parameter specified as DASD is entered, then the delay filing attribute is ignored. The request is handled as an immediate file.
- If the TPF system is running in a loosely coupled environment and the ZRTDM MODIFY command with the LOCKF parameter specified as PROC is entered, then delay filing is in effect and the updated copy remains in VFA on that particular processor.

Synchronized Delay Filing

Records are handled the same way as normal delay filing records described previously with the addition that a synchronized delay file candidate is filed when another processor requests the record and database consistency can be maintained if multiple processors update the record.

Immediate Filing
FILEC, FILNC, and FILUC macros are processed in the conventional way; the record is physically filed to DASD. Immediate filing occurs for delayed file records when the TPF system is not in NORM state.

Synchronized Immediate Filing
Records are handled the same way as normal immediate filing records described previously with the addition that database consistency is maintained if multiple processors update the record.

DASD Locking
Records that are to be held for exclusive use are fetched from DASD so that the external lock facility (XLF) can grant exclusive use to one processor. This is the default setting. Any record that will be altered should have this attribute.

Processor Locking
A record that is used in a read only capacity or is processor unique bypasses the XLF when the record is located in VFA. The VFA copy is fetched and no DASD I/O is performed.

Attention: Do not consider this option unless extremely high performance is warranted. Because the external lock facility is bypassed, the data integrity of the record may be compromised.

General Data Set Record
A special VFA candidacy condition exists for certain general data set (GDS) records. Index records that are associated with virtual storage access method (VSAM) data sets mounted to the TPF system are cached in VFA for performance reasons. VSAM record access requires an associated key lookup using an index data set; caching these index records eliminates the need for repeated accesses to DASD. The TPF system selects GDS record candidates internally; that is, the ZRTDM command cannot be used to select GDS record candidacy.

Maximum VFA Trip Value

The maximum VFA trip value provides a way for VFA to force an ECB to give up control when the ECB is a heavy user of VFA records.

In TPFGBL there are two values, &MXVFARQ and &MXVFARS. The default value for both is 50. When an ECB is initialized, the maximum VFA trip counter is set to the value in &MFVFARQ. Every time a file, find, or enter/back request is completed in VFA, the VFA trip counter is decremented by 1. When the counter reaches 0, the ECB is forced to give up control on the next WAITC or implied wait. The VFA trip value is then reset using the value in &MXVFARS.

Specifying VFA Buffer Ratios and Percentages

After a system IPL, you can enter the ZVFAC DEFINE command to modify VFA values for use in allocating main storage and building the VFA allocation chains. See *TPF Operations* for more information about the ZVFAC command.

If new VFA allocation values are used, perform a hardware IPL with the clear option after the TPF system is successfully cycled down from NORM state or successfully cycled to 1052 state.

If VFA has to rebuild VFA on an IPL, it will use the values stored in keypoint record A (CTKA) to allocate resources.

Restart Procedures

When VFA is fully operational and delay file mode records have been defined as VFA candidates, the VFA buffers can contain updated records that have not been filed to disk any time the TPF system is in NORM state. If a serious hardware or software error occurs, there is the potential that updated records can be lost.

For this reason, if you re-IPL the TPF system from NORM state, VFA restart determines if the VFA buffers are still valid and, if so, the updated buffers are immediately filed to DASD. The TPF system determines if VFA is valid since the last IPL. If VFA is valid, the TPF system does not clear the VFA area or allow a redefinition of the TPF system until VFA has been cleared. The TPF system only allows an IPL with the same user configuration. This allows VFA restart to file VFA buffers. Therefore, ensure that the system operator does not clear main storage before re-IPLing.

Note: An automatic re-IPL does not clear VFA storage.

If a CPU changeover is being considered following a serious error, consideration must be given to the updated VFA buffers that cannot be recovered if the TPF system is re-IPLed on a different CPU. Before changing CPUs, it may be worthwhile to try a re-IPL on the same CPU so the updated VFA buffers are filed.

Messages and Responses

The VFA commands provide the following operations:

- Define VFA buffer ratios, reserve chain sizes, and buffer reuse threshold values
- Enabling and disabling delay filing
- Display VFA status
- Display VFA buffer ratios, reserve chain sizes, and buffer reuse threshold values
- Indicate VFA utilization and efficiency
- Locate VFA residents
- Display VFA buffer usage
- Reset the RIAT control values.

The VFA commands are described in *TPF Operations*. Normal and error responses are described in *Messages (System Error and Offline)* and *Messages (Online)*.

Hardware Requirements

The following additional hardware is required by VFA:

- Additional main storage (depending on the use of the existing storage)
- The IBM 3990 Model 3 or later models with the multi-path lock facility (MPLF) installed if you are defining VFA synchronization candidates.

See the *TPF Migration Guide: Program Update Tapes* for more information about hardware requirements for the TPF system.

Record Caching

I/O performance is greatly enhanced when cache storage can directly access DASD records in contrast to the overhead required to perform physical DASD accesses to the DASD records. TPF support for DASD record caching is available with 2 features: The 3880 Record Cache RPQ and the 3990 Record Cache RPQ.

The IBM 3880 Storage Control Record Cache RPQ is a high performance record caching storage controller specifically designed to enhance TPF DASD I/O performance. The 3880 record caching support is limited to retentive data access. TPF support for the 3880 Record Cache RPQ is referred to as record cache (RC) support.

The IBM 3990 Storage Control Record Cache RPQ is a high performance record caching storage subsystem designed to further enhance TPF DASD I/O performance. The 3990 record caching support exploits the cache fast write and DASD fast write extended functions of the 3990 hardware as well as the retentive data access function of the 3880 Record Cache support. TPF support for the 3990 Record Cache RPQ is referred to as record cache subsystem (RCS) support.

This chapter gives you an overview of the record cache features of the 3880 Record Cache (RC) and the 3990 Record Cache Subsystem (RCS) RPQ. In addition, the TPF functions that can be used in conjunction with these RPQs are described. See the following hardware publications for complete operating information:

- *3880 Storage Control Record Cache RPQ Introduction*
- *3880 Storage Control Record Cache RPQ Description*
- *3990 Storage Control Introduction*
- *3990 Storage Control Planning, Installation, and Storage Administration*
- *3990 Storage Control Operations and Recovery Guide*
- *3990 Transaction Processing Facility Support RPQs*.

3880 Record Cache RPQ

This section provides an overview of the 3880 Record Cache RPQ and a description of the TPF functions you can use in conjunction with that RPQ.

Hardware

The 3880 Record Cache RPQ contains a large subsystem random access electronic storage (as many as 64 megabytes) used to store active DASD data for quick access. This means that frequently used DASD data can remain in the cache, reducing the number of repetitive DASD accesses to a single initial I/O operation. In addition, data stored in the cache can be accessed at a much higher speed than data stored in the DASD device.

Modes of Operation

The 3880 Record Cache RPQ operates in either record access mode or direct mode. All DASD devices attached to the storage director must operate in either mode. To directly support TPF, the 3880 Record Cache RPQ (operating in record access mode) stores single DASD records for accesses when requested by the TPF system. Direct mode can access data directly from the DASD devices.

Initial Program Load

One function of the TPF IPL is to initialize the record cache control unit and place the devices in record access mode, the normal operating state of 3880 attached record cache devices. However, initializing to record access mode does not take place for a general file IPL or an IPL of other than the first processor in a loosely coupled complex. When storage or hardware failures prevent initialization or ready status, or a device cannot be placed in record access mode, IPL processing notifies the operator and continues the IPL by placing the devices in direct mode.

If the system is IPLed before a cache allocation request completes, the request terminates and no allocation processing takes place. A warning message is issued to notify the operator and the IPL continues. It is the system operator's responsibility to insure that the pending allocation operation is completed to the 3880 Record Cache Control Units in the complex using either the prior (current) or new (target) allocation values.

DASD Processing and Error Recovery

TPF DASD support routines handle the new channel command words (CCW) for record cache as well as any new error conditions that may result. DASD processing CCWs are built by segment CJII from indicators set by IPL processing. Normally a buffered CCW string is built to access record cache DASD.

I/O Retry

On a buffered CCW I/O failure with unit check and command reject, the DASD program CCSONS retries the operation once using direct access CCWs. If the retry is successful, direct access CCWs are used to address the device until either a TPF IPL occurs or the device's status changes. If the retry is unsuccessful, a catastrophic error occurs.

Module Up/Down Processing

The ZMCPY DOWN command returns the record cache device to direct mode if it was in record access mode. This allows other processors not running TPF to access the device.

The ZMCPY UP command attempts to place the record cache device into record access mode. If the device is being placed online on an available and initialized record cache control unit, an attempt is made to place the device into record access mode. If this attempt fails, the device is placed in direct mode and an error message is issued.

If the device is the first device to be placed online for the record cache control unit then the cache must be made ready before the device is put into record access mode and initialized. If any failures occur, the device is left in direct mode.

The processing of a general file mount and dismount is similar to module up and down.

Data Collection and Data Reduction

Performance statistics are collected for 3880 Record Cache Control Units and devices attached to the control units. The ZMEAS command is used to offload the collected data from each device. These statistics are displayed on the reports generated by data reduction.

Restrictions

Nonretentive data is not supported by this RPQ. Loader general files cannot be attached to this RPQ. Devices are not sharable among subsystems, but are sharable among processors running the same subsystem.

Command Description

The TPF commands that support 3880 Record Cache operations are:
- ZBUFC ALLOCATE
- ZBUFC ALLOCATE IMPLEMNT
- ZBUFC ALLOCATE DISPLAY
- ZBUFC STATUS.

You are not required to modify the record cache, but you can adjust the ratios for block sizes in keypoint 0 from the default values.

Note: Throughout the TPF publications, the term *ratio* refers to the hardware term *weighted values*.

In addition, you can monitor data block usage and control unit status as explained later in this section.

Specifying Block Size Ratios

You can partition the record cache into the 3 TPF record sizes (381, 1055, and 4096 bytes) by assigning block size ratios. Defaults for all 3 ratio values are set in keypoint 0 at one, dividing the cache equally among the TPF block sizes. In addition, one full track buffer is defaulted in keypoint 0.

You can change the block size ratios by using the ZBUFC ALLOCATE command. The allocation is a 2 step process:

1. First use the ZBUFC ALLOCATE RC381-*rr*,RC1055-*rr*,RC4096-*rr*,RCBUF-*bb* command to ***specify*** the target allocation values.
2. Next use the ZBUFC ALLOCATE IMPLEMNT command to ***implement*** those values. These values will then be applied to all the record cache control units in the complex.

The ZBUFC ALLOCATE DISPLAY command displays the current and target cache allocation settings for the specified record cache control unit.

Monitoring Control Unit Status

The ZBUFC STATUS command lets you monitor the status of a control unit. This display informs you of device operating modes, storage and interface status, and the storage size allocated to each TPF block size.

For example, record access mode status information can help you determine which devices to take offline. Placing these devices online again requires the ZMCPY UP command and each record caching device should be brought online in record access mode. If the device being placed online is the first device on a specific control unit, then TPF issues an initialization sequence for the control unit before placing the device into record access mode.

3990 Record Cache RPQ

This section provides an overview of the 3990 Record Cache RPQ and a description of the TPF functions you can use in conjunction with that RPQ.

Hardware

The 3990 Record Cache RPQ is a high performance record cache subsystem that provides improved throughput for DASD. The hardware contains 2 storage clusters, each with 2 storage paths. There are as many as 8 channel attachments per cluster and an inboard dynamic path selective (DPS) array. The hardware has subsystem storage with as many as 256 megabytes of storage and a nonvolatile storage (NVS) with as many as 4 megabytes of storage. The hardware can be configured as 2 subsystems splitting the DPS, cache, NVS, and storage paths and sharing the channels (device level selection (DLS) configuration); or as a single subsystem with 4-path devices (device level selection enhanced (DLSE) configuration).

As with the 3880 Record Cache Control Unit, a subset of data stored on DASD can also be stored in the cache. This data can then be accessed from the cache instead of DASD to improve the I/O response time of the system. Additionally, the extended function capabilities of cache and DASD fast write provide the benefits of cache hits to write operations. A write operation indicating fast write data is done at cache speed and does not require an immediate transfer of data to the DASD surface. This data is written directly to the cache or nonvolatile storage and is available for later destaging to the DASD surface.

Modes of Operation

The 3990 Record Cache Subsystem supports the following modes of access:
- Direct
- Track caching
- Record access mode.

Direct mode provides access to the attached devices as if they were attached via a noncached nonbuffered subsystem. Direct mode is normally used for activities that require direct access to the device, such as formatting or diagnostics, or for data accesses when the cache is either not operational or not available to the subsystem.

Track caching mode maintains full or partial track images in the cache according to usage algorithms. Multiple record data transfer operations (that is, full track reads or writes) use track caching algorithms.

Record access mode maintains individual record images in the cache according to usage algorithms. Single record data transfer operations use record caching algorithms.

Programmable Options

The 3990 Record Cache Subsystem RPQ supports the use of both CKD and ECKD channel command chains. Record mode I/O operations can specify the following access characteristics:
- Bypass cache (direct mode)
- Retentive
- DASD fast writes
- Cache fast writes.

Note: I/O operations that do not indicate record mode will operate in direct (bypass cache) mode.

The differences between retentive, cache fast write, and DASD fast write, all of which are caching accesses, are as follows:

- Retentive places the data in the cache, writes the data to the DASD surface, and then presents device end status.
- DASD fast write places the data in the cache and in the nonvolatile storage and then presents device end status. The data is written to DASD when space is needed in the nonvolatile storage or in the cache, or when directed by a host CCW request.
- Cache fast write places the data in the cache and then presents device end status. The data is written to DASD only when cache space is needed or when directed by a host CCW request.

TPF Record Cache Subsystem (RCS) Support

System Installation Procedure (SIP)

The system installation procedure supports the 3990 Record Cache Subsystem RPQ by:

- Reserving space in #IBMM4 for the file copy of the record cache subsystem status table (SSST).
- Inserting the maximum configured 3990 record cache subsystem SSID in keypoint 0 from the IODEV macro parameters.
- Providing a SYSTC macro switch when the system is generated as having RCS devices from the IODEV macro parameter information.
- Building the record caching attribute information in the RIAT table from information in the RIATA macro specifications.

  Note: If no caching attribute is specified for a given record ID, the default caching attribute is retentive.

Extended RIAT Support

The Extended RIAT table is used to contain the record caching attributes associated with a particular record ID. Table 3 summarizes these attributes and their functions.

Table 3. Record Caching Attributes

Attribute	Function	Description
RET	Retentive Access	The specified record ID is placed in the volatile control unit cache and on the DASD surface when a file-type macro is issued.
CFWS	Cache Fast Write Access (Simplex Write)	The specified record ID is placed in the volatile control unit cache when a file-type macro is issued and the cache is available. If the cache is not available, the record will be written directly to the DASD surface. A single write is issued to the prime module only.
CFWD	Cache Fast Write Access (Duplex Write)	The specified record ID is placed in the volatile control unit cache when a file-type macro is issued and the cache is available. If the cache is not available, the record will be written directly to the DASD surface. A duplexed write is issued to both the prime and the duplicate modules.
DFW	DASD Fast Write Access	The specified record ID is placed in the cache and the nonvolatile storage when a file-type macro is issued. If the cache is not available, or the nonvolatile storage is not available, the record is written directly to the DASD surface.
NO	Bypass Cache	The specified record ID is not a candidate for caching and all I/O requests for this ID result in the record being read/written directly from/to the DASD surface, thus bypassing caching for this record ID.

Note: All read operations are done using the cache fast write attribute in order to detect possible data loss conditions at the earliest possible time.

Initial Program Load

During IPL, each DASD attached to a 3990 Record Cache Subsystem Control Unit is marked as a caching device. In addition, IPLB indicates whether or not a DASD is a real-time module.

Unit check conditions caused by conflicting device information on the status tracks of the DASD attached to the 3990 Record Cache Subsystem Control Unit may result in acquiring access to the DASD by forcing the subsystem to re-establish its global status information.

Unlike the 3880 Record Cache Control Unit, IPL of the loader general file attached to a record cache subsystem control unit is supported.

System Initializer

The system initializer allocates storage for the record cache subsystem status table (SSST). Both a memory copy and a file copy of this table are maintained. This table contains the status of each record cache subsystem control unit in the system. The system initializer also allocates storage for the AET (asynchronous event table) that is used to monitor asynchronous I/O request operations. Additional processing is performed to set up the dump table pointers and control table address pointers for system use.

System Restart

The RCS restart component of TPF system restart brings all 3990 Record Cache Subsystem Control Units and their attached devices to full caching capability, initializes cache slot allocations, and detects and reports possible data loss conditions since the last TPF IPL. This function also builds the record cache subsystem status table data structures.

DASD Processing and Error Recovery

Write I/O operations to RCS devices are controlled by the record caching attributes specified in the RIAT table for a specific record ID. The CCW chains are dynamically modified on each I/O operation as required to specify the caching attribute as indicated in the RIAT table.

The following restricted use macros result in using the retentive caching attribute as the default:

FILSC	File a single file record
FINSC	Find a single file record
FLSPC	File a special record
FNSPC	Find a special record
FDCTC	File data chain

Cache Fast Write Processing: Records with the fast write caching attribute can be indicated as cache fast write simplex or duplex. When cache fast write is indicated, all read operations are done from the prime module only. This insures consistent data in the event of a cache failure. Write operations will be done to the prime only if CFW simplex was specified; otherwise, the writes are performed to both the prime and duplicate modules.

Module Up and Down Processing: When a module down operation completes, the module is taken offline. All data in the cache is discarded because the module is no longer considered current. If you want the module being taken offline to be current, make sure a ZBUFC FILE command was previously completed to the device. This will insure that fast write data is destaged to the DASD surface. The module up function initializes devices that are attached to the record cache subsystem control units.

Asynchronous Event Processing: Some control I/O operations associated with the cache may take a long time to complete. These are I/O operations associated with the Set Subsystem Mode (SSM) and the Perform Subsystem Function (PSF) CCWs. A facility is provided whereby a long running operation can be started and allowed to run asynchronously with other I/O activity. Actual completion of the request is signalled some time later by the hardware via an attention interrupt. TPF associates the completion of the operation with the original request by using the asynchronous event table. This feature is only valid for I/O operations issued through the use of the FDCTC macro.

Error Recovery Methodology

TPF recognizes the following error situations:

1. Unable to enable cache capability.

   This condition results when a hardware failure is detected that precludes TPF from using a caching resource.
2. Potential data loss detection.

   This condition is reported during RCS restart and can result from the movement of DASD from a record cache subsystem or a change in cache fast write ID since the last TPF IPL.
3. Hardware failure while running.

   This condition is reported via unit check/sense information or from receipt of a state change interrupt.

In all cases, operator notification is provided indicating the error and the affected RCS subsystem(s).

State Change Processing: Changes in the status of resources in the hardware are signalled through a state change interrupt presented to the TPF system. This state change notification results in the TPF system requesting latest subsystem status and taking appropriate actions depending on the status change detected.

I/O Queue Thresholding: TPF considers any RCS to be running in a degraded mode of operation if any loss of caching capability occurs. If this happens, TPF provides an I/O queue thresholding mechanism to monitor the total subsystem queue depth while an RCS is running in a degraded state. If the total queue depth exceeds an RCS limit computed from user-specified information, a user exit is invoked and actions are taken as specified by the user exit routine.

Data Collection/Data Reduction

Caching performance statistics are maintained in the 3990 Record Cache Subsystem. These statistics are collected for the record cache subsystem control units and attached devices. The ZMEAS command is used to offload the collected data from each device. These statistics are displayed on the reports generated by the data reduction package.

Restrictions

- A DLS configuration is only supported for attachment of a single 3380 DASD device string.

- A DLSE configuration is only supported for attachment of a single 3390 DASD device string.
- The dual copy facility of the 3990 subsystem is not supported.
- In a loosely coupled environment, TPF only supports a single path to a device.

  Note: The TPF system supports multipathing with the concurrency filter lock facility (CFLF).
- The record cache subsystem SSID must be unique for all subsystems in the complex.

Command Description

The commands that support the 3990 Record Cache Subsystem (RCS) Control Units are:

- ZRTDM DISPLAY
- ZRTDM MODIFY
- ZBUFC ALLOCATE
- ZBUFC ALLOCATE IMPLEMNT
- ZBUFC ALLOCATE DISPLAY
- ZBUFC FILE
- ZBUFC ENABLE
- ZBUFC STATUS
- ZBUFC THRESHLD
- ZBUFC PINNED DISPLAY
- ZBUFC PINNED DISCARD
- ZBUFC SETCACHE
- ZBUFC MAP

Specifying Record Caching Attributes

The ZRTDM DISPLAY command can be used to display the record caching attributes for specified record IDs or for the short- and long-term pool overrides.

The ZRTDM MODIFY command can be used to modify the record caching attributes for specified record IDs or for the short- and long-term pool overrides.

Specifying Block Size Ratios

You can partition the record cache subsystem into the 3 TPF record sizes (381, 1055, and 4096 bytes) by assigning block size ratios. Also, you can assign full track slots to be used for buffering.

You can reallocate cache storage by following these 2 steps:

1. First use ZBUFC ALLOCATE RCS381-*pp*,RCS1055-*pp*,RCS4096-*pp*,RCSBUF-*tt* to ***specify*** the target allocation values.
2. Next use ZBUFC ALLOCATE IMPLEMNT to ***implement*** those values. These values will then be applied to all the record cache subsystem control units in the complex.

The ZBUFC ALLOCATE DISPLAY command can be used to show the current and the target cache allocation ratios.

Disabling Fast Write Caching Functions

The ZBUFC FILE command allows you to destage all modified data in the cache and nonvolatile storage to the DASD surface. In addition, all fast write caching capability is disabled to prevent any further modified data from entering the cache. The ZBUFC ENABLE command is used to re-enable fast write caching capability.

Enabling Caching Functions

The ZBUFC ENABLE command lets you re-enable all caching capability of an RCS Control Unit and its attached devices. This message should be used after a ZBUFC FILE was issued.

Displaying Control Unit Status

The ZBUFC STATUS command displays the status of the RCS Control Unit and attached devices.

The ZBUFC STATUS RCS command displays internal TPF status information associated with the RCS support package.

Displaying or Changing I/O Queue Threshold Value

The ZBUFC THRESHLD command lets you display or modify the current I/O queue threshold value. This value is used to calculate the overall threshold value used when the TPF system is monitoring the I/O queues for an RCS running in degraded mode.

Displaying Pinned Data Report

The ZBUFC PINNED DISPLAY command lets you display the pinned data for a given record cache subsystem attached device. Pinned data is data in the cache or nonvolatile store indicated as having been unable to be written to the DASD surface.

Discarding Pinned Data

The ZBUFC PINNED DISCARD command allows you to discard all pinned fast write data associated with a specific DASD device from the 3990 Record Cache Subsystem cache and nonvolatile storage.

Use this command only for recovery from cache related error conditions where removal of the pinned data is required to recover fast write caching function for the device or 3990 Record Cache Subsystem.

Set Cache Operating Modes

The ZBUFC SETCACHE command allows you to alter the 3990 Record Cache Subsystem caching status for the various caching resource elements for the purposes of recovery cache capability following certain cache related errors.

Use this command only for recovery from cache related error conditions.

Attention: Use this command carefully; data loss from cache or nonvolatile storage is possible.

Displaying Map Report

The ZBUFC MAP command lets you display the relationship between an RCS control unit and its attached devices. The report indicates the RCS SSID associated with a particular device or the symbolic device address range(s) associated with a particular RCS SSID.

Processing Differences between the 3990 Model 3 and the IBM Enterprise Storage Server

TPF record cache subsystem (RCS) support was introduced with the 3990 Model 3 Record Cache RPQ. With the introduction of the 3990 Model 6, subsequent generations of DASD controllers that are record cache capable provide the original 3990 Model 3 Record Cache RPQ functions as standard product features. To maintain backward compatibility with TPF RCS support as well as to provide a migration path for TPF users, these controllers provide a TPF mode setting in the

Vital Product Data (VPD) area of the controller. This setting enables the controller microcode to maintain the original Record Cache RPQ API for TPF. This concept is further provided with the IBM Enterprise Storage Server (ESS) by allowing the definition of 3990 Model 3 TPF logical subsystems.

From a TPF software perspective, communicating with DASD attached to newer versions of the 3990 is logically the same as communicating with DASD attached to the original 3990 Model 3 controller. However, the internal processing related to the caching and handling of various I/O operations has changed. These changes simplified and streamlined internal processing by removing a number of the processing options that were available to the TPF system in the original record cache implementation while taking over the management of other aspects of the cache that previously required user action.

Operationally, the TPF system maintains the intent of the original 3990 Model 3 Record Cache RPQ implementation, but the effects of the processing actions in the ESS have changed. The trade-off to the TPF system is to maintain compatibility with the existing code while allowing for a migration path to the ESS for TPF users.

The following table summarizes the differences in the internal processing between the 3990 Model 3 and ESS as they relate to the TPF system.

Table 4. Processing Differences between the 3990 Model 3 and ESS

Caching Attributes	
3990 Model 3: The record ID attribute table (RIAT) specifies caching attributes on an individual I/O basis. Valid caching attributes are as follows: • Cache data in the volatile cache using the cache fast write (CFW) attribute • Cache data in the nonvolatile cache using the DASD fast write attribute • Cache data in the volatile cache and write it to the DASD surface using the retentive write attribute • Bypass caching data records by writing directly to the DASD surface.	**ESS:** By default, all written data is handled as fast write data and is written to both cache and nonvolatile storage (NVS): • Data is only placed in cache if the I/O requests fast write. • Data is placed in both cache and NVS if the I/O is a normal cache replacement. TPF I/O requests with one of the fast write attributes will function as originally implemented. Attributes requesting to bypass caching result in the use of the cache but the records age out of cache by means of accelerated least recently used (LRU) lists.
Enabling Cache and Nonvolatile Storage (NVS) Differences affect record cache subsystem (RCS) restart actions, which ensure all cache capability is fully enabled and ZBUFC command processing, which allows user control in the preparation for cache allocation and cache recovery action.	
3990 Model 3: Users have two levels of control: • Volatile and nonvolatile caching elements associated with a specific device • Volatile and nonvolatile caching elements associated with all devices at a subsystem level. Each caching element allows for user control to enable or disable at the device or subsystem level.	**ESS:** Controls associated with options of the set subsystem mode (SSM) channel command are at a higher level. While the I/O API associated with SSM is still supported, the operations at the control unit are largely accepted but ignored. For example, only three SSM parameter options are functional on the ESS: • Activate CFW for the subsystem • Deactivate CFW for the subsystem • Force deactivate DASD fast write for devices.

Table 4. Processing Differences between the 3990 Model 3 and ESS (continued)

Record Cache Allocation	
3990 Model 3: Cache memory must be partitioned into record and multi-path lock facility (MPLF) slots in varying proportions for the standard TPF record sizes. Users must allocate the cache and the associated ratios appropriately for optimum usage.	**ESS:** Partitioning of the cache for record and MPLF lock slots is managed by the control unit and is maintained for optimum usage based on processing patterns. TPF I/O API operations to set and query the cache allocation are maintained and still reflect desired ratios as requested by the ZBUFC commands. The actual allocation, however, is managed internally and dynamically by the control unit.
Performance Statistics	
3990 Model 3: Cache statistics are reported for each TPF record size (381, 1055, or 4096 bytes) for each device. This information is collected by TPF data collection processing for offline data reduction reporting and could be used to tailor the cache allocations in the most optimum way.	**ESS:** Manages the cache allocation dynamically and, as a result, only one set of cache statistics is maintained for each device. The statistics are normalized for the TPF data record sizes and reported in terms of the TPF 1055-byte record size.
ZBUFC Commands See *TPF Operations* for more information about the ZBUFC commands.	
ZBUFC ALLOCATE Use this command to set the desired cache allocation ratios for both record and lock space.	
3990 Model 3: Users specify cache and lock space allocations for optimum performance.	**ESS:** Actual allocations and cache partitioning are managed dynamically by the control unit. For compatibility reasons, the specified TPF allocation values are returned to the TPF query allocation requests.
ZBUFC ALLOCATE IMPLEMNT Use this command to apply the allocation changes that were specified using the ZBUFC ALLOCATE command.	
3990 Model 3: Allocation changes are applied to the control units that are not currently initialized with the new allocation values.	**ESS:** Accepts and ignores the TPF request to change the actual cache allocations in use in the control unit, although the desired TPF ratios will be returned on allocation queries from TPF.
ZBUFC FILE Use this command to destage existing modified cache data and prevent additional accumulation of modified data in cache or NVS in preparation for the hardware reconfiguration or system shutdown.	
3990 Model 3: I/O operations requesting data records be written with one of the fast write caching attributes results in the data being written to both the cache and the DASD surface.	**ESS:** Only I/O operations requesting data to be written with the CFW attribute are affected and result in the data being written to both the cache and the DASD surface.
ZBUFC ENABLE Use this command to enable all the caching functions for one or all of the 3990 RCS control units in the complex.	

Table 4. Processing Differences between the 3990 Model 3 and ESS (continued)

<table>
<tr><td><b>3990 Model 3:</b>

I/O operations requesting data records be written with one of the fast write caching attributes results in data being written to cache or NVS, and subsequently, destaged to DASD as cache space is needed.</td><td><b>ESS:</b>

Only I/O operations requesting data to be written with the CFW attribute are affected and result in the data once again being written to the cache, and subsequently, destaged to DASD as cache space is needed.</td></tr>
<tr><td colspan="2"><b>ZBUFC SETCACHE</b>
Use this command to change the cache resource states for the 3990 RCS to perform cache error recovery actions.</td></tr>
<tr><td><b>3990 Model 3:</b>

Allows you to perform various cache recovery operations. The syntax allows you to enable or disable cache elements identified at a device or subsystem level.</td><td><b>ESS:</b>

The following functional processing options are affected for this command:
• Activation/deactivation of CFW
• Force deactivation of DASD fast write for a device.

All other options are accepted and ignored at the control unit.</td></tr>
</table>

Part 3. Utilities

This part contains information about the following database utilities:
- Capture and restore
- Database reorganization (DBR)
- Recoup
- Real-time disk formatter
- FACE driver and offline interface (DFAD)
- TPF transaction services
- TPF collection support.

Capture and Restore

The information that is stored on the online disk files represents a considerable investment by the user. The integrity of this data is vital to system operation.

The capture and restore utility serves as the vehicle for system fallback. It allows you to capture to tape all of the system variable online data. If a serious hardware, software, or operator-induced failure occurs, you can restore all or selected parts of the online data files to their state at a predetermined point in time.

Note: With TPF transaction services processing, the capture and restore utility will bring the TPF database back to a consistent commit state when possible.

Capture and Restore Processing Overview

The capture and restore utility is controlled by the ZFCAP and ZFRST commands, respectively. The commands support options to start, abort, pause, restart, add or delete tape devices, and request status. A single purpose, multiple-entry concept is used to ensure concurrently overlapping I/O operations. The messages are verified and sent to the participating processors where the appropriate routines are started. It is important that you identify the participating processors before the capture function starts. Once the function starts, you cannot add more processors to the participation list. Use the ZPROT command to add and assign processors for the capture and restore utility. See *TPF Operations* for more information about the capture and restore utility commands.

Automatic tape mounting is suspended for capture and restore, except for devices used for exception recording and keypoint capture and restore. If capture processing starts on a device that is enabled for automatic tape mounting and the device contains an ALT tape, the suspend processing dismounts the ALT tape without unloading it.

Capture has a message analyzer to:
- Verify messages
- Determine the participating processors
- Synchronize exception recording on the processors
- Set up the necessary tapes
- Start capture processor activity.

The restore message analyzer validates the message and starts the restore processor activity.

Special purpose programs are called to provide specific elements of the capture and restore utility. Notable routines handle start/continue/EOJ, read/write, restart, abort/pause, tape add/delete, status, exception recording/logging, security, and error recovery.

Data filed out as part of normal capture (as opposed to exception recording) is written to general tapes referred to as *capture tapes*. These tapes can be subsequently read by the restore function. Tape drives and tape drive pairs for use by capture or restore are listed on the ZFCAP or ZFRST command. Additional drives or pairs can be added during capture or restore. Each tape drive or pair of drives can be used to capture 1 module at a time. Channel and control unit utilization limits determine how many of the assigned tape drives may be in use by capture at one time. See “Capture Processing” for more information about channel and control unit utilization.

One disk module can require more than 1 tape for its capture. If the module is being captured to a tape pair, at the end of the first tape, data continues to be filed out to a tape on the second drive in that pair (called the *alternate drive*). The drives in a tape pair flip-flop in this way until the entire disk module is captured. If the tape drive was assigned to capture as a single tape drive, and not as a tape drive pair, then capture first attempts to find another single drive (that was assigned to capture) that is not currently in use. If no other drive is available, the system prompts the operator to mount a standby capture tape.

Restore works in a similar way, with all tapes for a given module loaded to a single drive or to a pair of drives.

Real-time tapes are used for exception recording. Exception recording tapes must be mounted by the operator for all active processors before capture can be started. To restore the records on these tapes, the tapes used for all processors active at the time of capture must be mounted for a single processor so that the records can be read from tape and filed in the correct order.

Capture Processing

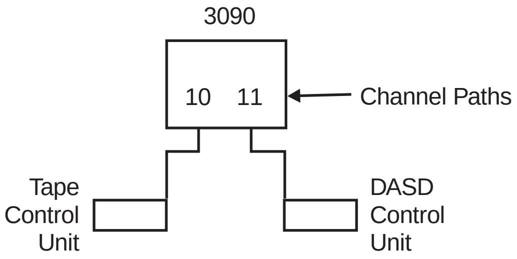
Either all or selected online disks can be captured. Duplicate copies of duplicated records on each disk are not captured. Many modules can be preserved simultaneously. The modules to be captured and their respective tapes should be chosen to obtain the maximum possible channel separation. The total number of disk captures that are active at any one time depends on a particular system’s configuration and the user-specified limits on load balancing. This number, however, can be changed by a command. To improve operator efficiency, the capture program selects a tape on which to capture a specific disk, assigns a symbolic tape name to each tape, and causes the tape to be internally mounted.

During disk capture, you control the load balancing across channel paths and control units by limiting the maximum number of captures that can occur simultaneously on a:
- Channel path for DASD or tape
- Device control unit for DASD or tape.

Use of a channel path and control unit are limited to the values set for the DASDCU, DASDCH, TAPECU, and TAPECH parameters. For example, if a channel group is handling the maximum amount of captures for tapes, no additional captures will be allocated to the tape devices that are serviced by that group until the number of captures are reduced.

Capture sets the value for DASDCU to 1; you cannot change this value. This ensures that no group of channel paths or control units is overused and keeps DASD and tape module queues at manageable lengths. You can change the value for DASDCH, TAPECU, and TAPECH with the ZFCAP CHANGE command.

A typical configuration of DASD and tapes channel paths has the DASD and tape control units on separate paths.

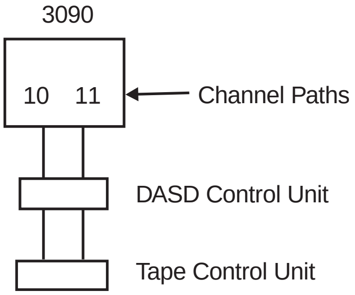


The maximum values for the preceding figure are:

```
DASDCU: 1  TAPECU: 1  DASDCH: 1  TAPECH: 1
```

In other words, 1 DASD can participate in a capture under any given control unit and only 1 DASD can be started on a channel path, or CHPID, at a time; similarly for tape.

If an installation takes advantage of multipathing, the channel paths can run through the same control units to provide the additional pathway flexibility.

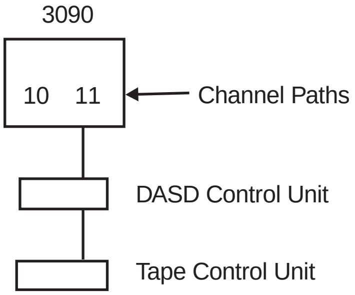


The maximum value for this installation is:

```
DASDCU: 1  TAPECU: 1  DASDCH: 1  TAPECH: 1
```

This is the same as the previous configuration, but in this case multipathing definitions have been put into the IOCP generation providing more flexibility.

There is a special consideration when a channel path (CHPID) is shared between a tape device and a DASD on the same processor. The DASDCH or TAPECH values must include any DASD or tape running on the same channel path. If the number of captures is incremented, the second kind of capture (DASD or tape) requested is not started until the first is completed. This is shown in the following figure:



The maximum value for this installation is:

```
DASDCU: 1  TAPECU: 1  DASDCH: 2  TAPECH: 2
```

Notice that the number of captures on the channel path for both DASD and tape is the number of control units on that path that can possibly be expected to be running capture.

The relationship between the DASDCU and DASDCH, and the corresponding TAPECU and TAPECH parameters provides a means for balancing capture resource requirements across numerous devices, control units, and channel paths. For example, in Figure 6, a single channel path or CHPID is shown with 2 DASD control units (C1 and C2) and 3 devices (D1, D2, D3). DASDCU is set to 1. This means that only a single device will operate at a time under a given control unit. The DASDCH parameter allows you to specify how many devices can be run under a single CHPID.

For example, when DASDCU equals 1 and DASDCH equals 1, you can capture device D1. Devices D2 and D3 are not available for capture at the same time. If you leave DASDCU equal to 1 but set DASDCH equal to 2, you can capture 2 devices, D1 and D2, at the same time. You cannot capture device D3 at the same time because DASDCU is 1. DASDCU is restricted to 1 for performance reasons. (An example of TAPECU > 1 appears later.) If you leave DASDCU equal to 1 but now set DASDCH equal to 3, 3 devices can be captured at the same time on the CHPID. The configuration to do this is for another control unit to appear below C2, call it *C3*, with the third device attached to it. Device D3 in this example would not participate in the capture at the same time as D1 (because DASDCU equals 1).

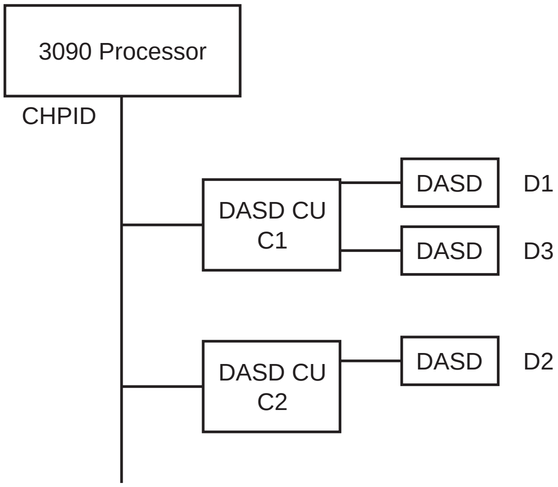


Figure 6. Capture and Restore Load Balancing Using DASDCU and DASDCH

There are configuration considerations for the techniques used for balancing capture loads for the following IBM 3990 models:
- IBM 3990 Model 2 with the limited lock facility (LLF)
- IBM 3990 Model 3 with LLF running record buffer emulation.

The LLF static switch may cause even and odd addresses to be placed on different channels. For example, if your configuration defines device addresses 4E0, 6E1, 4E2, 6E3, and so on, the even addresses are placed on channel 4 and the odd addresses are placed on channel 6. However, for correct load balancing, all the device addresses of these control units must have the same channel address. For example, the addresses in the previous example can be defined as 4E0, 4E1, 4E2, 4E3, and so on. This is ***not*** a concern for the IBM 3990 Model 3 running record cache.

Load balancing is directly related to site resource requirements and the degree of multipathing found in the configuration. The configuration is described during Input/Output Configuration Program (IOCP) generation. See *ES/9000, ES/3090 Input/Output Configuration Program User's Guide and ESCON Channel-to-Channel Reference* for more information about IOCP generation.

The TAPECU and TAPECH parameters function similarly. The TAPECU parameter is not restricted to one with tape, however.

The top part of Figure 7 on page 60 shows 1 CHPID connecting a processor with a single tape control unit. Under this control unit there are 4 tape devices. If TAPECU equals 1 and TAPECH equals 1, only 1 of these tape devices can participate in capture at a time. If TAPECU equals 4 and TAPECH equals 1, only 1 tape can participate at a time. If TAPECU equals 4 and TAPECH equals 4, all 4 tape devices can run capture at the same time.

The bottom part of this figure shows 4 CHPIDs connecting the single tape control unit. This is a multipathing configuration. Using the multipathing aspect you can set TAPECU equal to 4 (to allow 4 tape devices to operate under the same control unit) but set TAPECH equal to 1 (thereby limiting the number of tape devices operating at the same time on the CHPID to 1). The multiple channel paths to the same control unit satisfy this restriction, but still allow 4 tape devices to be run at the same time. So, the two parts of this figure provide the same results, which is 4 tape devices participating in capture at the same time.

The first part shows the TAPECH parameter set to allow more tapes on the CHPID (TAPECH=4). In the second part the additional CHPIDs are used to run those same 4 tape devices while restricting each CHPID to a single tape device. Load balancing helps to balance the capture resource requirements.

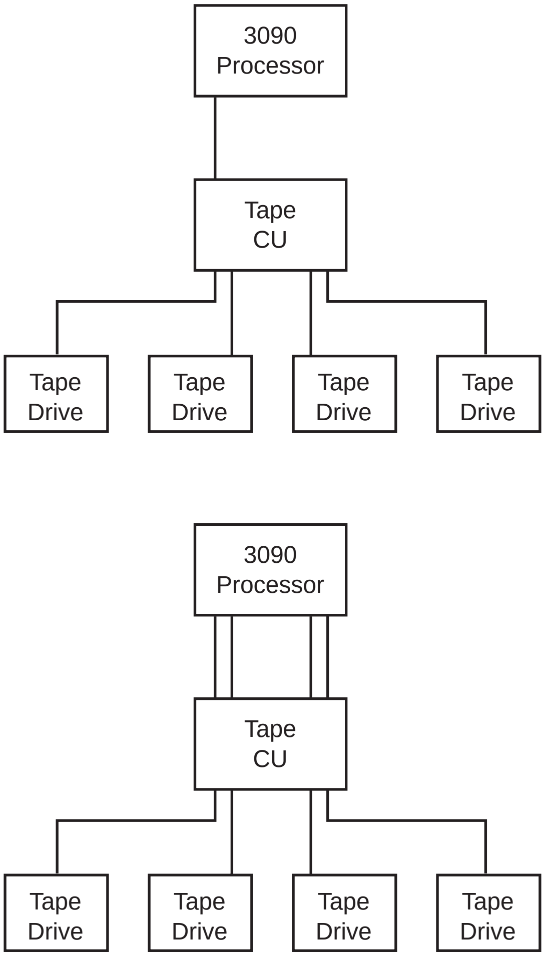


Figure 7. Capture and Restore Load Balancing Using the TAPECU and TAPECH Parameters

The capture functions are independent entities for each subsystem in a multiple subsystem environment. Multiple subsystem capture can occur simultaneously. However, from a performance standpoint, it should be limited to a single subsystem per processor.

There are 2 options available when starting capture. The first option causes the capture of all online disk packs. The second option results in the capture of specified modules. For a capture all (the ZFCAP ALL command) request, specify tape devices from all tape channels. The capture tapes must be pre-initialized and placed on the tape drive for each participating processor and made ready. The capture function assumes that all tape drives specified to it as available are always made ready with a capture tape. The capture function selects a tape drive to achieve the best possible channel separation, mounts the tape internally, handles switching to multiple reels if necessary, and causes the tapes to be rewound and unloaded when completed. The capture function selects the module to be captured to a particular tape based on disk channel and control unit separation.

In a loosely coupled environment, capture is started from a single processor. Processor participation is determined by active entries in the processor resource ownership table (PROT) that are assigned before starting capture with the ZPROT command. The individual option of file capture is limited to a single processor. See *TPF Operations* for more information. Before capture starts, the input message is verified to determine the following:

- At least 1 capture tape device was specified for each participating processor.
- All participating processors are in utility (UTIL) state or higher.
- Exception recording (XCP) tapes are mounted on all active processors, whether they are participating or not.
- There is an active PROT entry for the initiating processor.

When the previous conditions are met, the capture function is started in the designated processors and exception recording (XCP) is started in all active processors. Once started, each processor proceeds independently in selecting modules for capture so as not to interfere with the modules already in progress by other processors. When a participating processor exhausts all selection possibilities, a PROCESSOR COMPLETE message is displayed and the capture function ends for that processor.

In loosely coupled and multiple database function (MDBF) systems, the control unit tables are processor-shared, though the values for DASDCU and TAPECU are not shared. These values are held in processor-unique, capture working keypoints. This means that apparent inconsistencies can occur. For example, if TAPECU equals 5 on CPU B and TAPECU equals 10 on CPU C, 10 captures can be started on a control unit, even though processor B sets the limit to 5. In effect, the highest DASDCU or TAPECU value across all of the processors prevails. The same apparent inconsistency can occur for subsystems also. For example, if TAPECH equals 5 in the basic subsystem (BSS) and TAPECH equals 10 in SS2 (a non-BSS), 10 captures can be started on the CHPID tape despite the limit in the BSS. There are no restrictions to prevent subsystems from sharing CHPIDs or control units.

Exception Recording

Capture is designed to run in a real-time environment. Therefore, records can be updated on disk files that have already been written to the capture tape. When file capture is started, an indicator is set for the control program, which examines the address of each record that is written to disk file when this indicator is on. If the address is behind the current capture location, a copy of the record is written to an exception recording (XCP) tape. You have the option of not performing exception recording certain record types. For example, you may choose to ignore short-term record updates in order to reduce the exception recording (XCP) tape load on the TPF system. This is done by turning off these record exception recording indicators in the record ID attribute table (RIAT). Exception recording is not stopped automatically when capture is completed in order to allow the captured data to reflect the state of the online files at later time. Discontinue exception recording by entering a command. If an IPL-restart occurs, the capture function itself must be restarted by a console order but exception recording automatically continues.

After a capture of all modules, and while exception recording is still in progress, individual modules can be recaptured and exception recording will continue on all modules. This is done by using the individual mode of capture and can be done as many times as necessary, but ***only*** following a capture of all modules. A capture of all modules cannot be started while exception recording is in progress.

The exception recording and logging tapes are real-time tapes. Therefore, tape drive utilization can be improved by mounting an alternate (ALT) tape instead of a standby output tape.

Record Logging

Record logging allows you to log to tape, at any time, all updates to user-specified record types on file. These tapes are the real-time logging tapes. Logging is started and discontinued by entering a command.

Note: If logging is active when capture is started, the logging tapes will also be used as the exception recording tapes.

When multiple subsystems exist, record logging is started on a subsystem level. The real-time logging tapes may be unique or shared by the different subsystems.

In a loosely coupled environment, the start and stop logging functions affect all active processors and thereby require active logging tapes on all processors for the function to start.

If an IPL-restart occurs, logging continues automatically.

Capture of Keypoints

The keypoint capture function is provided to capture the system keypoint records, tape label directory (TPLD) records, and tape label mask records (TLMR) to the KPC tape. Keypoint capture is started automatically at the end of the file capture function when the stop exception recording message is displayed following a capture ALL request. The subsystem keypoints for all generated processors are captured to the KPC tape. Keypoints that are shared between subsystems are captured only when keypoint capture is processing in the basic subsystem (BSS). Configuration-dependent keypoints (keypoints 1, 6, and I) will not be captured. Keypoint capture can also be started by command input.

Restore Processing

There are 3 phases to restore processing:
- Phase 1 restores the capture tape records.
- Phase 2 restores logging and exception recording tape records.
- Phase 3 restores keypoints.

In phases 1 and 2, duplicated records are restored on both the prime and duplicate modules, if any, if the following conditions exist:
- A restore ALL function is started.
- The duplicate restore option is selected.
- Both the primary and duplicated modules were selected for restore.
- A selected module is fully duplicated.

If none of these conditions exist, duplicate tracks are not restored on either the primary or duplicate modules.

The 3 phases for restore are described in more detail in the following sections.

Restore Generic Capture Tape (CAP)

Phase 1 restores the capture tape records. When a restore is required, you have three options for the restore of the capture tape file (CAP):
1. Restore all online disk packs.
2. Restore only specified modules.
3. Restore all records for a given address range.

The optional parameter (D) may be added to the end of each message to indicate the duplicate restore option. In options 2 on page 62 and 3 on page 62, duplicated records are restored whether or not both the primary and duplicate copies lie in the area to be restored.

You may also specify the optional FZ and BP parameters. The FZ parameter means that zeroed records are to be filed. The BP parameter means that reel sequence checking will be bypassed.

If more than 1 tape is required, as in the case of a restore ALL request, tape drives should be selected from as many different channels as possible. This will minimize restore time. The drives specified in the command should be separated by slashes (/). The mounted tapes should be for disk modules on the maximum number of different channels. Subsequent restore tapes should be placed on the available drives in the same way.

If multireel tapes are to be restored, tape device pairs should be used to minimize delays because of tape rewind. The 2 drives must be on the same channel and control unit. The first reel should be mounted on the active drive and the next reel on the alternate drive. Subsequent reels would be mounted in a flip-flop fashion. When the rewind of the next-to-last reel is completed, that drive should be made ready with the first (or only) reel of another module to be restored (if any modules still remain to be restored).

You only need to ready the tape drives. Restore ensures that the tapes are mounted and will rewind and unload them when finished. However, restore assumes that all drives that are made available to it have CAP tapes on them. If a tape mount error occurs, the drive should be made ready with the correct input tape but you should not enter the tape mount command.

In a loosely coupled environment, as previously described, file restore is started from a single processor. Processor participation in the restore function is determined by active entries in the PROT table. See *TPF Operations* for more information about the ZPROT command.

Once started, each processor proceeds independently restoring from the tape devices assigned to that processor. As in capture, the processors that are to participate in the restore must be active when the function is started. No provisions are available to allow more processors to participate once the function is started. The selective module restore function is limited to a single processor. The restoration of data can be to a single disk or a number of disks concurrently. As with capture, the number of disks to be operated on at any one time depends on the system configuration.

Image-related records will not be restored when records associated with TPF images are bypassed. Records that are bypassed include:
- Core image restart area (CIMR) records
- Keypoint records
- IPL area records
- Keypoint staging area records
- E-type program area records
- Program history area records
- Image pointer records (CTKX)
- Any record types that have RESTORE=NO coded on the RAMFIL macro, including #XPRG records containing (ISO-C) C programs.

You can restore these records to a module with the ZIMAG COPY command or by using one of the loaders (general file loader or auxiliary loader).

In addition, the tape label directory (TPLD) records and tape label mask records (TLMR) will not be restored because they are actively used during the restore function. These records are captured on the KPC tape by the keypoint capture function, and you can restore these records to a module using the ZFRST KPT command.

Restore Logging and Exception Recording (XCP) Tapes

Phase 2 restores logging and exception recording tape records.

The records that are logged to the real-time logging and exception recording tapes can be restored to all modules, a specific module, or an address range in a module or encompassing many modules. Time parameters are required that specify where logging restore will start and where it will stop, determined by time-stamp records on the restore tapes. The time parameter for exception recording (XCP) specifies where XCP restore will start. This time is generally the time when capture was started.

The restore of logging and exception recording (XCP) tapes in a loosely coupled environment requires that you mount tapes for all processors that were active at the time of capture. This function is not shared by the participating processors but can be started in any one of them.

Restore Keypoints

Phase 3 restores the keypoint records.

Keypointing is prevented while a restore is in progress to ensure the integrity of the captured database. While keypointing is prevented by restore, updates are still allowed to keypoints that are not captured (see *TPF Operations*). Because these keypoints cannot be restored, data corruption during restore is not a concern. Data corruption of restored keypoints (including globals) is also not a concern during a module restore because a subsequent IPL is not expected. Therefore, keypointing is prevented only by the following input messages:

- ZFRST CAP ALL
- ZFRST XCP ALL
- ZFRST KPT
- ZFRST RESTART (if after ZFRST CAP ALL).

Keypointing remains disabled until you perform the subsequent mandatory IPL. If you perform the IPL before the logging records are restored, another IPL is required to enable keypointing. IPL as soon as possible after the restore to prevent possible data corruption by the time-initiated keypoint update routines. In a loosely coupled environment, the complex must be collapsed down to 1 processor before you re-IPL. This ensures that the proper initialization of the keypoint records takes place at system restart.

Note: Keypoint B is not fully restored; the VFA and lethal utility switches are not restored.

Capture Considerations

Depending on your operational schedule, file capture can be done in two ways:

- Real-time online capture.

- Non-real-time online capture. This has the advantage of limited or no file interference, and no exception records are written. This method cannot be used for a continuously operating system.

It is recommended that you preform a capture daily in order to minimize the data loss if an irrecoverable failure occurs. The optimum time to capture, from the point of view of file regeneration, is immediately following file maintenance. In addition to the daily capture, it is recommended that files are captured:
- Before and after fixed-file reorganization
- Before and after file pool reallocation
- Before major program updates.

Record Logging

Record logging can be used as:
- A complete file recovery means against hardware failure but at a cost of additional system load and hardware requirement during peak periods of operation
- A test tool to verify the updating of certain record types during program modifications or additions
- A means of data collection for additional analysis.

Keypoint Capture

The keypoint capture function is normally used after file capture to obtain the latest copy of the keypoints.

Restore Considerations and Procedures

As mentioned earlier, you can restore all modules (referred to as a *total restore*) or a selected area of a file or single module (referred to as a *partial restore*).

A total restore is necessary for the following conditions:
- Destructive online testing of the system
- Major system software failure affecting vital records
- Major hardware failure in which both prime and duplicate copies of some records are destroyed
- Major operator error causing destruction of vital system records.

A partial restore is necessary for a hardware failure; for example, damage to a module or a bad drive causing writes to the module that are not valid.

Note: With TPF transaction services, partial restores will not necessarily produce a consistent commit system.

For a damaged track, a file copy should be carried out to another module and the unreadable records repaired or initialized under coverage programmer control. File recoup can be used as a tool for detecting chains in this process that are not valid. This procedure allows the TPF system to remain online for normal activity during the repair process. See “Recoup” on page 103 for more information about recoup.

Restore Procedures

This section describes the procedures for a:
- Total restore without a database
- Total restore with a database

- Partial restore.

Total Restore without a Database

This procedure is for restoring a TPF system (basic subsystem (BSS) or subsystem) that has no database.

To restore the captured data records to their respective locations on the online files, prepare all the system disks so that the TPF system can be IPLed, cycled to UTIL state, and restored. To do this:

1. Format all disks according to the system configuration.
2. Do a full load of the general file. See *TPF System Installation Support Reference* for more information about the loaders program.
3. IPL the general file and cycle the system to 1052 state.
4. Deactivate and initialize the tape label directory (TPLD) records and the tape label mask records (TLMR) with the ZTLMR command.
5. Create tape labels for data pilot tapes (SDF) and restore tapes (KPC) using the ZTLBL command.
6. Activate tape label directory records and tape label mask records using the ZTLMR command.
7. Load the following pilot tapes:
   - Data records
   - SCK.

   See *TPF System Installation Support Reference* for more information about loading tapes.
8. Load the SNA definitions using the ZNOPL command.
9. IPL the online module and cycle to utility (UTIL) state.
10. Restore the system keypoint records, the tape label directory (TPLD) records, and the tape label mask records (TLMR) from the KPC tape by entering the ZFRST KPT command.
11. Follow the procedures for a total restore in General Total Restore Procedures.

See *TPF Operations* for information about all the commands.

Total Restore with a Database

To do a total restore of packs that have incorrect data, follow these steps:

1. Cycle the system or desired subsystem to UTIL state.
2. Follow the procedures for a total restore in General Total Restore Procedures.

General Total Restore Procedures

After following the procedures outlined in “Total Restore without a Database” or “Total Restore with a Database” continue as follows:

1. Restore all the disk files from the latest set of capture tapes.
2. Re-IPL the TPF system.
3. Restore the exception recording (XCP) tapes.
4. Restore the system keypoint records, if you did not do this procedure already in “Total Restore without a Database”.
5. If the system is loosely coupled, collapse the complex to 1 processor.
6. Re-IPL the TPF system and cycle up to utility (UTIL) state.
7. Restore the LOG tapes if logging was active during capture.
8. Reconcile the file pool counts with the ZRFPC command.

9. If you did not select RCB records or short-term pool records for exception recording during real-time capture, they must be initialized to delete incomplete transactions and references to short-term pool addresses. This is done by using the ZRCBI command.
10. Because transactions that were completed since capture are determined by the automatic switching of the RTA at the conclusion of exception recording, it may be desirable to run file regeneration if compatible with any logged data.
11. You can bring the system online by entering the ZCYCL NORM command. GFS restart will start and update the FPDR. Run recoup to verify the file pool directory records.

Partial Restore

For a restore of 1 or multiple modules, follow these steps:

1. Format the disk modules according to the system configuration. See “Real-Time Disk Formatter” on page 117 for more information.
2. Cycle the system to 1052 state because records that are needed in the restart schedule will be duplicated on other modules. In this case, it is assumed that the restart records are duplicated records.
3. Remove the RTA tape by entering the ZTOFF command for the RTA tape.
4. IPL the system. If the new pack is the prime module, IPL from the duplicate module.
5. You will receive a message requesting you to mount the RTA tape. Enter messages to take the modules to be restored offline.
6. Mount the RTA tape.
7. The system will now come up to 1052 state. Enter the messages to bring the requested modules back online. This will cause duplicate updating to start automatically.
8. Cycle up the system to utility (UTIL) state.
9. Start the restore for the requested modules.

This procedure can be followed for more than 1 module as long as a pack and its corresponding duplicate are not both down.

Timing

During capture, entire disk tracks are read at a time by using chained channel control words, including both count and data fields. A search command is not used to check the transfer, but a software check is made when the read is completed to ensure that the proper track has been read. This permits the reading of a track to start after the first address marker is sensed, making rotational delay minimal. One tape record is written for each track of DASD records that is read. The sequence of reading DASD tracks runs from head 0 through the last head on cylinder 0, followed by head 0 through the last head on cylinder 1 and so on for the device that is being captured.

Restore also uses chaining to read a capture tape and write 1 DASD track at a time. The tracks are restored in the sequence previously described for capture. The average rotational delay is about half a disk track. On a restore with the duplicate option, the prime tracks are written to the module concurrently with the duplicates on the duplicate module. If the entire system is being restored, the duplicates are also restored.

Assumptions and Conditions Relative to Timing Example

To simplify timing calculations for a capture and restore operation, certain conditions are assumed and described as follows. These values can be changed to match individual conditions.

1. It is assumed for this example that a single 3380-D DASD has 50% total tracks allocated and the system is fully duplicated. Because capture copies only the primary copies of duplicated tracks, only half the modules in a system are captured.
2. 10% of allocated primary DASD tracks each contain ten 4KB records, 30% of DASD tracks each contain thirty 1055-byte records, and 60% of DASD tracks each contain fifty-three 381-byte records.

   This DASD information for a 3380-D is summarized as follows:

Record Size	Records per Track	Percent Allocated
381 bytes	53	60%
1055 bytes	30	30%
4096 bytes (4KB)	10	10%

3. Tape calculations for 3480s and 3490s assume:
   - 18 tracks
   - 38 000 BPI
   - 2 millisecond start/stop time
   - 4.5 MB per sec instantaneous data rate
   - 150 meters tape length.

   More exact tape timing, using effective data rates expressed in megabytes (MB) per second for 3380-D tracks written out as tape blocks containing 381-, 1055-, and 4096-byte (4KB) record sizes follows:

Device Type	381-byte Track	1055-byte Track	4096-byte (4KB) Track
3480	2.35MB per sec	2.5MB per sec	2.5MB per sec
3490	3.0MB per sec	3.0MB per sec	3.0MB per sec

   The effective data rate values shown include start/stop time and are based on capturing a single disk module at a time to a single tape drive with negligible interference from other system activity.

4. The amount of time required for switching the tape in a multicartridge capture or restore has not been factored into the estimates.
5. Tape record size per 3380 DASD track =

```
(( nbr records per track ) * ( nbr bytes per record )) + 508
```

   This calculates to the following tape block sizes for the 3380-D to 3480 tape example:

DASD	Tape Block Sizes
381 bytes	20 702
1055 bytes	32 162
4094 bytes (4KB)	41 468

6. The maximum main storage used by each entry is calculated as follows:

```
Total Main Storage per Entry =
  ( 2 * (( nbr 381 records per track  *  381 record size )
      + ( nbr 1055 rcd per trk * 1055 rcd size )
      + ( nbr 4KB rcd per trk * 4KB rcd size ))  )
+ ( nbr work blks * blk size )
```

At any given time the main storage utilization per entry is the amount of work space used plus the space required for the track size of the device that is being processed.

For the devices that are being supported, the main storage values are shown as follows:

Device	Work Space	381 Track Space	1055 Track Space	4KB Track Space	Total
3380	8KB (4 blk)	40KB (106 blk)	62KB (60 blk)	80KB (20 blk)	190KB
3390	8KB (4 blk)	41KB (110 blk)	68KB (66 blk)	96KB (24 blk)	213KB
9345	8KB (4 blk)	35KB (94 blk)	56KB (54 blk)	80KB (20 blk)	179KB

7. The *system pause time-out factor* is the maximum amount of time allowed for an entry to complete after a pause has been requested. Entries exceeding this limit are timed out and aborted. The minimum amount for this field should be at least ***the maximum time*** required to capture or restore a module on any in-use device type ***plus*** an amount required if the module does not have exclusive use of a channel.
8. To determine the minimum and maximum rewind and unload time for all tape devices used by capture and restore, do the following:

```
(Minimum time = maximum rewind and unload time for the slowest tape device)
                                +
(Time needed by the operator; for example, time for mounting the tape)
```

See the RESCAP macro in *TPF System Generation* for the default maximum rewind and unload time.

```
Tape time per track  =
    ( tape block size per track ) / ( effective data rate )
```

Capture Timing Estimates

The following timing estimates will assist you in calculating your capture and restore time requirements. The assumptions used should be carefully examined to determine if they are correct for a particular system.

Note: The longer of the disk read and tape write times determines the time required to capture the assumed configuration.

Disk Time Calculations

The following table and calculations show the time required to capture data from the various 3380 DASD models supported by the TPF system.

Average rotational delay is calculated as follows:

```
Average Rotational Delay =

    ((rotation time / nbr 381 rcds per track) / 2) * (percentage of 381-byte tracks)
 +  ((rotation time / nbr 1055 rcds per track) / 2)) * (percentage of 1055-byte tracks)
 +  ((rotation time / nbr 4K rcds per track) / 2)) * (percentage of 4K-byte tracks)
```

device type	nbr cylinders/module	cyl-cyl seek time (ms)	average seek time (ms)	nbr tracks to capture	percentage of module used	381-/1055-/4096-byte track ratio	nbr tracks / module	Average Rotational Delay (ms)	rotation time (ms)
3380D,J	885	3,2	15,12	6,638	50%	60/30/10	13,275	0.19	16.7
3380E	1770	3	17	13,275	50%	60/30/10	26,550	0.19	16.7
3380K	2655	2	16	19,913	50%	60/30/10	39,825	0.19	16.7
3390I	1113	1.5	9.5	8,348	50%	60/30/10	16,695	0.19	14.1
3390II	2226	1.5	12.5	16,695	50%	60/30/10	33,390	0.19	14.1
3390III	3339	1.5	13.5	25,043	50%	60/30/10	50,085	0.19	14.1
9345I	1440	1.5	10	10,800	50%	60/30/10	21,600	0.19	11.2
9345II	2156	1.5	11	16,170	50%	60/30/10	32,340	0.19	11.2

```
Disk time per module  =

    ( nbr tracks to capture * ( rotation time + avg rotational delay ))
 +  ( average seek time )
 +  ( cyl-cyl seek time * (( nbr cyl * pct used) - 1 ))
```

For example:

```
3380-D time/mod = 6,638(16.7 + 0.25) + 15 +  3(885*0.5 - 1)  =  117.8 sec.
```

Tape Time Calculations

The following calculations establish the tape time per captured module. The tables that were shown previously provide supporting information to verify the arithmetic.

Note: Tape drives use the EDR that is expected for the block size that is being written (32 158-byte blocks for 1055-byte record tracks, 20 701-byte blocks for 381-byte record tracks, and 41 468-byte blocks for 4096-byte record tracks), and the number of records written is the sum of the large, small, and 4KB tracks to capture from the particular device.

Tape time per module = ((nbr bytes per pack) / (EDR for tape record size))

```
For 381-byte blocks on 3480:
    3,982 tracks * 20,701 bytes/track / 2.35 MB/sec. = 35 sec.
For 1055-byte blocks on 3480:
    1,991 tracks * 32,158 bytes/track / 2.5 MB/sec.  = 26 sec.
For 4KB blocks on 3480:
      664 tracks * 41,468 bytes/track / 2.5 MB/sec.  = 11 sec.
                                                    ___
                               Total tape time      = 72 sec.
```

The conclusion from these calculations is that the 3380-D DASD example can be captured to 3480 tape in 117.8 seconds, the longer of the 2 timings (72 seconds for tape time and 117.8 seconds for DASD).

Restore Timing Estimates

Single Module Restore Timing Calculations

The following table and calculations show the disk time required to restore 3380 devices. Search time is calculated as follows:

```
Search time =  .5 * rotation time + tape read time - disk write time
```

Tape read time varies for 381-, 1055-, and 4096-byte (4KB) record tracks. This table shows the values as calculated from the effective data rates (EDR) for each record size for both tape drives. Tape time/module is calculated the same as for capture. Restore time is the greater of either tape time/module or disk time/module.

device type	nbr cyl/module	nbr 4k trks	nbr large trks	nbr small trks	average seek time (ms)	rotation time (write time) (ms)	cyl-cyl access time (ms)	tape rd time/4096-byte track (ms)	tape rd time/1055-byte track (ms)	tape rd time/381-byte track (ms)	
3380	885	664	1992	3983	15	16.7	3	16.6	12.9	.8	(3480)
3390I	1113	835	2504	5009	9.5	14.1	1.5				
3390II	2226	1670	5009	10017	12.5	14.1	1.5				
3390III	3339	2504	7513	15026	13.5	14.1	1.5				
9345I	1440	1080	3240	6480	10	11.2	1.5				
9345II	2156	1617	4851	9702	11	11.2	1.5				

Disk time per module =

```
Factor
   I     ( total nbr tracks to restore ) * rotation time

  II     + (( tape rd time per 4K rcd trk - write time
                  + ( 0.5 *  rotation time  )) * nbr 4K non-dup trks )

 III     + (( tape rd time per 1055 rcd trk - write time
                  + ( 0.5 *  rotation time  )) * nbr 1055 non-dup trks )

  IV     + (( tape rd time per 381 rcd trk - write time
                  + ( 0.5 *  rotation time  )) * nbr 381 non-dup trks )

   V     +  average seek time

  VI     +  ( cyl-cyl seek time * ( nbr cyl per mod  -  1 ))
```

Nonduplicate Restore:

```
   FACTOR             I                     II                                III
3380 time/mod  =  6638 * (0.0167) + (0.0165 - 0.0167) + (0.0083 * 664) + (0.125 - 0.0167) + (0.0083 * 1992)
                 (0.0085 - 0.0167) + 0.0085 * 3982) + 0.015 + (0.003 * (443 - 1))
                            IV                      V           VI

               = 167.3 sec
```

Duplicate Restore:

```
   FACTOR             I                      II                                  III
3380 time/mod  = 13276 * (0.0167) + (0.0165 - 0.0167) + (0.0085 * 1328) + (0.125 - 0.0167) + (0.0085 * 3986)
                 (0.0085 - 0.0167) + 0.0085 * 7964) + 0.015 + (0.003 * (443 - 1))
                              IV                      V            VI

               = 333.3 sec
```

Tape Volume Estimates

The volume of tape produced is calculated as follows:

```
Length in inches   =   (( nbr bytes transferred ) / ( BPI rate ))
                     + (( IRG size ) * ( nbr records written ))
```

where:

- BPI is bytes per inch. The possible values for the BPI rate are:

38K	for a 3480 or 3490
38K2	for a 3490E.

- IRG is the inter-record gap, which is the space between the physical tape blocks. The value for the IRG is:

0.08 inches	for a 3480 or 3490.

The following are sample calculations for several 3380-to-tape captures.

Note: These samples do not take into account any improvements possible with the Increased Data Recording Capability (IDRC) feature available with the 3480 and 3490.

```
277,045,053/38K + .08(11616) = 8,220 in. = 685 feet (3480 38K BPI)
```

Capture and Restore Keypoint Record

The capture and restore keypoint (BXAXF) record is used for control and communication by the capture and restore utility. Fields in the capture and restore keypoint that must be changed to suit the user’s system can be established with the system initialization package (SIP). The SIP macro and macro parameters that will initialize the field are noted after each BXAXF description as follows:

SIP input = (macro keyword name) macro name

The SIP macros that are related to the capture and restore utility are RESCAP, DDCCAP, and LOGCAP. See *TPF System Generation* for more information about coding these macros.

Keypoint Fields

Table 5 shows a description of each of the fields in the keypoint record. Unless otherwise stated, SIP initializes these fields to X'00'.

Table 5. Keypoint Fields

Name	Description
BXXLAB	This tag is used to refer to the program record number and demand counter that are used in lieu of an alphabetic record ID because the capture and restore keypoint is maintained as a program. Initialization: Dynamic via assembly and link edit.

Table 5. Keypoint Fields (continued)

Name	Description
BXXNAM	This tag is used to refer to the program stamp field. When initially loaded, this field contains the name of the keypoint program (BXAX or BXX1). After running online, the working keypoint will change to BXAM, the program name of the keypoint update program. Initialization: Dynamic via assembly.
BX1IND	This field contains 8 1-bit indicators used by the capture function to maintain major package status.
BX2IND	This field contains 8 1-bit indicators used by the restore function to maintain major package status.
BX3IND	This field contains 8 1-bit indicators used by both capture and restore to maintain major package status.
BX4IND	This field contains 8 1-bit indicators used by the restore function to maintain status of the XCP/LOG tapes being restored.
BX5IND	This field contains 1-bit indicators used by both capture and restore to maintain status of the participating processors.
BXXCPTP	This field contains the number of active XCP/LOG tapes for restore.
BXXSID	This field contains the symbolic processor ID.
BXXIID	This field contains the internal processor ID.
BXXMAX	This field controls the maximum number of entries that can be created by capture or restore, therefore controlling the maximum number of modules that can be captured or restored simultaneously. Initialization: See “Timing” on page 67.
BXXEBR	This field contains the LNIATA of the input message that started the current function.
BXXOID	This field contains the CPUID of the input message that started the current function.
BXXCIP	This field is used to maintain a count of the capture or restore entries currently in progress.
BXXCES	This field is used by the restore selection routines as a count of entries that are in the process of being started but have not yet been started.
BXDDCTC	This field is used to unhook the DDCT during capture and restore operations, therefore providing a core chain to the DDCT.
BXDDCB	This subfield of BXDDCTC contains the address of the core block that contains the DDCT. A load from this location sets the base register for DDCT DSECT usage.
BXXSSI	This field contains the ID of the subsystem in which the capture and restore function is running.
BXXDTE	This field is initialized by the restore function with the current date in the form of *ddmmm* (day/month). This value is taken from TPF global field @U1DMT.
BXXTME	This field is used by the restore function as the starting time of the restore. It is initialized by restore from control program field CMMLST, referred to with a CINFC macro.
BXXLDT	This field is initialized by the Logging program. It contains the last date a label record has been written to the logging tapes in the form *ddmmm* (day/month). This value is taken from global field @U1DMT.

Table 5. Keypoint Fields (continued)

Name	Description
BXXLTM	This field is initialized by the Logging program. It contains the last time, in hours and minutes, that a label record has been written to the logging tapes, in the form *hhmm*. This value is taken from the CMMLST field, referred to with a CINFC macro.
BXSTRD	This field is initialized by the restore function with the start date input in the XCP/LOG restore input message after it has been converted to binary form by segment CDTA.
BXSTRT	This field is initialized by the restore function with the start time input in the XCP/LOG restore input message after it has been converted to binary form by segment CDTA.
BXSTPD	This field is initialized by the restore function with the stop date input in the LOG restore input message after it has been converted to binary form by segment CDTA.
BXSTPT	This field is initialized by the restore function with the stop time input in the LOG restore input message after it has been converted to binary form by segment CDTA.
BXXTOT	This field is used to maintain a count of the total modules to be captured or restored.
BXXDON	This field is used to maintain a count of the total modules that have completed capture or restore. When this field is equal to BXXTOT, the entire capture or restore operation is completed.
BXACPS	This field contains the system pause time-out factor, which is the maximum amount of time allowed for an entry to be completed when a pause is requested. If an entry does not complete processing in this amount of time, it is timed out and will end. Initialization: The minimum number of seconds used to initialize this field should be the maximum time required to capture or restore a module of any in-use device type. This time should take into consideration the additional time required by a module not having exclusive use of a channel. SIP input = (CAPTO) RESCAP.
BXRWD	This field contains the maximum rewind and unload time for all tape devices that can be used by capture and restore. An amount of time equivalent to that specified by this field is allowed to elapse after this tape is rewound before making it available for use. Initialization: The minimum value should be equivalent to the maximum rewind and unload time for all tape devices on the system. SIP input = (TPREW) RESCAP.
BXIOTIME	This field contains the minimum amount of elapsed time between capture I/Os. At the completion of each I/O, the clock value just before the I/O is compared to the current clock value. If the difference is less than the value in BXIOTIME, the I/O entry defers because capture I/O requests are completing too quickly. The entry continues to DEFER until the required amount of time has elapsed.
BXDASDCH	This field contains the maximum number of DASDs capable of running per channel path. Initialization: This field is initialized to 1.

Table 5. Keypoint Fields (continued)

Name	Description
BXDASDCU	This field contains the maximum number of DASDs capable of running per control unit. Initialization: This field is initialized to 1.
BXTAPECH	This field contains the maximum number of tape drives capable of running per channel path. Initialization: This field is initialized to 1.
BXTAPECU	This field contains the maximum number of tape drives capable of running per control unit. Initialization: This field is initialized to 1.
BXANDV	This field contains the number of devices in the system. Initialization: This field is initialized to the number of devices in the CDT (L'BXADTS/L'BXACDT).
BXAMID	This field contains the maximum size of the module interval table. Initialization: This field is initialized to the length of field BXDINT.
BXASDT	This field contains the size of an entry in the device control table (see field BXACDT). Initialization: This field is initialized to the length of field BXACDT.
BXXPGM	This field is used by the restore function as a save field for the type of restore: CAP, XCP, LOG, or KPT.
BXXERR	This tag refers to the fields containing the cumulative error counts for tape errors (BXXTAP), disk track errors (BXXTRK), disk record errors (BXXRCD), and a spare field of 2 bytes.
BXXTAP	This field contains a cumulative count of all uncorrectable tape read errors found during restore. Capture does not use this field.
BXXTRK	This field contains a cumulative count of the uncorrectable disk track read errors found since the last capture or restore start request.
BXXRCD	This field contains a cumulative count of the number of individual record errors found since the last capture or restore start request. The counts in this field and BXXTRK are mutually exclusive.
BXXTTA	This field is used by the restore function to accumulate the number of tapes currently available for selection.
BXXNSL	This field is used as a count of TDCT slots that are in use during restore. Therefore, it serves as an index to the next TDCT slot that is available to the selection routines.

Keypoint Tables

The following tables are defined in the capture/restore keypoint record (BXAXF).

Tape Device Control Table (TDCT)

The tape device control table is used to keep an inventory and status of the tapes assigned to capture and restore.

Table 6. Tape Device Control Table

Name	**Description**
BXTDCT	This tag defines the tape device control table (TDCT). This table is used to maintain an inventory and status of all tape drives made available to capture or restore by the operator through a start or tape add request. Initialization: This tag is initialized with a length attribute of the TDCT table size.
BXTDCTF	This tag defines an entry in the tape device control table. There should be as many entries in the TDCT as are required by multiplying BX1MAX (the maximum number of entries allowed active) by 2. This allows for uninterrupted processing during rewind. Initialization: The length attribute of this tag is the size of an entry in the TDCT.
BXCUD1	This subfield of BXTDCTF is used to maintain the tape device hardware address input by the operator.
BXCUDA	This subfield of BXTDCTF is used to maintain an alternate tape device hardware address input by the operator. This field is used only by restore in multireel restores.
BXCUD1S	This subfield of BXTDCTF contains the symbolic tape name associated with the tape hardware addresses in BXCUD1 and BXCUDA. When capture and restore provide internal tape mounting and switching, this is the tape name that will be used. Initialization: This field is set to BX*x*, where *x* is set to A and incremented through the alphabet for as many entries as exist in the TDCT.
BXSTAT1	This subfield of BXTDCTF is used as bit indicators to maintain the status of the tape during the capture or restore operation.

LOG/XCP Tape Definition Table (LXTD)

The LOG/XCP tape definition table (LXTD) defines the logging and exception tapes available to capture and restore.

Table 7. Tape Definition Table (LXTD)

Name	**Description**
BXTAPS	This label refers to the LOG/XCP tape definition (LXTD) table. This table defines the tapes available to the capture and restore utility for logging and exception recording. Initialization: The length attribute of this label is equal to the table length.
BXTTIS	This label refers to an item in the LXTD table. Each item contains a tape name (last character), an ECB count field, and a module range associated with this entry. Initialization: The length attribute of this label is equal to the item length.
BXTNAM	This subfield of BXTAPS contains the last character of the tape name for this entry (for capture, the first 2 characters are RT; for restore, the first 2 characters are XA). Initialization: The allowable characters for tapes in this table are Y, X, W, V, U, and T respectively. Any item not used is initialized to X'00'. SIP input = (RTT thru RTY) LOGCAP.

Table 7. Tape Definition Table (LXTD) (continued)

Name	**Description**
BXTNEN	This subfield of BXTAPS indicates whether the tape for this entry is in use for capture or restore.
BXTLOM	This subfield of BXTAPS contains the low symbolic module number assigned to this tape for XCP/LOG capture and restore. Initialization: The initialization of this field and BXTHIM depends on what use will be made of the data captured to these tapes. Normally, each item should be initialized so that the load will be evenly spread across all tapes defined in the table. However, other considerations may dictate other initialization schemes. If an item is unused, this field will be initialized to X'00'. SIP input = (RTT–RTY) LOGCAP.
BXTHIM	This subfield of BXTAPS contains the high symbolic module number assigned to this tape for XCP/LOG capture and restore. Initialization: See BXTLOM.

In-Progress Table (IPT)

The in-progress table (IPT) maintains the status of reads and writes that are currently in progress.

Table 8. In-Progress Table (IPT)

Name	**Description**
BXINPT	This tag refers to the in-progress table. This table is used to maintain the status of currently in-progress read/write entries. Initialization: This tag has a length attribute equal to the size of the in-progress table.
BXINPF	This tag refers to an entry in the in-progress table. Initialization: This tag has a length attribute equal to the size of an entry in the in-progress table.
BXMODA	This subfield of BXINPF contains the module number of the module being captured or restored.
BXCUD2	This subfield of BXINPF contains the hardware address of the tape currently being captured to or restored from.
BXCUD2S	This subfield of BXINPF contains the symbolic name being used to address the tape on the address contained in BXCUD2.
BXTMES	This subfield of BXINPF contains the decimal time that the capture or restore entry began.
BXTMESB	This subfield of BXINPF contains the binary time that the capture or restore entry began processing the current reel of tape.
BXDEVT	This subfield of BXINPF contains the device type indicator of the disk device being captured or restored. This indicator is identical to the device type indicator in this device’s CDT entry (BXADTI).
BXSTAT2	This subfield of BXINPF is an end-of-reel switch used by the restore function to indicate when a new reel is being processed.
BXCDTI	This subfield of BXINPF is used by the restore function as an index to the correct device control table (CDT) for the entry in progress.

Table 8. In-Progress Table (IPT) (continued)

Name	Description
BXRRNO	This subfield of BXINPF is used to maintain the reel number of the current reel during a capture and restore operation. If a restart occurs, this will be the reel where capture and restore will restart.
BXTAP	This subfield of BXINPF contains the count of the uncorrectable tape errors that restore has found during the processing of this entry. When this entry is completed, this counter will be added to the cumulative tape error counter, BXXTAP.
BXTRK	This subfield of BXINPF contains a count of uncorrectable track errors found by capture or restore during the processing of this entry. When this entry is completed, this counter will be added to the cumulative track error counter, BXXTRK.
BXRCD	This subfield of BXINPF contains a count of the individual disk record errors found by capture or restore during the processing of this entry. When this entry is completed, this counter will be added to the cumulative disk record error counter, BXXRCD. The counts in BXTRK and BXRCD are mutually exclusive.
BXXSP2	This tag is used to refer to the current and restart locations (cylinder and head) for this entry during a capture operation.
BXXCCL	This tag refers to the current module location (cylinder and head) during a capture operation. Initialization: This tag has a length attribute of 4.
BXXCC	This subfield of BXINPF contains the current cylinder number of the module that is in progress during capture processing.
BXXCH	This subfield of BXINPF contains the current head number of the module that is in progress during a capture operation.
BXXCCL1	This tag refers to the restart module location (cylinder and head) during a capture operation. Restart during capture occurs from the beginning of a tape reel so that the cylinder and head value saved here reflect the beginning of the reel that is being processed. Initialization: This tag has a length attribute of 4.
BXXCC1	This subfield of BXINPF contains the restart cylinder number of the module that is in progress during a capture operation.
BXXCH1	This subfield of BXINPF contains the restart head number of the module that is in progress during a capture operation.
BXXBSN	This subfield of BXINPF contains the tape block sequence number of the first tape block (record) written to the current reel for capture. It contains the tape block sequence number of the last tape block (record) written to the previous reel for restore. If a restart occurs, this number is used to begin sequencing records on the restart reel.
BXXPID	This subfield of BXINPF contains the ID of the processor in which the capture or restore is running.
BXXEXT	This refers to the module extent table of extents that will not be restored.

Device Type Control Table (CDT)

The device type control table (CDT) contains descriptions of each DASD device that supports capture and restore.

Table 9. Device Type Control Table (CDT)

Name	Description
BXADTS	This tag refers to the device type table (CDT). This table contains parameters describing the characteristics of each supported device type. Initialization: This tag has a length attribute equal to the size of the CDT.
BXACDT	This tag refers to an entry in the Device Control Table (CDT). There is an entry for each supported device type. The entries do not need to be in any specific order and spare CDT items can exist (they must be initialized to all hex zeros). Initialization: This tag has a length attribute equal to the length of a CDT entry item.
BXADTI	This subfield of BXACDT defines the device type indicator for this device. This device type indicator is used primarily as a branch vector indicator in capture or restore segments that are device dependent. Initialization: The device type indicator is set to the equate for the specific device type (DEVA for device A, DEVB for device B, and so on). SIP input = DDCCAP macro.
BXASRD	This subfield of BXACDT defines the number of small records per track for this device type. Initialization: This field is initialized by SIP by setting it equivalent to the correct equate for the number of small records per track for this device type (CSONSRA for device A, CSONSRB for device B, and so on).
BXALRD	This subfield of BXACDT defines the number of large records per track for this device type. Initialization: This field is initialized by SIP by setting it equivalent to the correct equate for the number of large records per track for this device type (CSONLRA for device A, CSONLRB for device B, and so on).
BXA4RD	This subfield of BXACDT defines the number of 4KB records per track for this device type. Initialization: This field is initialized by SIP by setting it equivalent to the correct equate for the number of 4KB records per track for this device type (CSON4RA for device A, CSON4RB for device B, and so on).
BXASRL	This subfield of BXACDT defines the small record length for this device type. Initialization: This field is set to the equate for small record size for this device type (#SBSZE for all devices).
BXALRL	This subfield of BXACDT defines the large record length for this device type. Initialization: This field is set to the equate for large record size for this device type (#LBSZE for all devices).
BXA4RL	This subfield of BXACDT defines the 4KB record length for this device type. Initialization: This field is set to the equate for 4KB record size for this device type (CPL4C2 for all devices).

Table 9. Device Type Control Table (CDT) (continued)

Name	Description
BXAHDC	This subfield of BXACDT contains the maximum addressable head number for this device type. Initialization: This field is set to one less than the equate that defines the number of heads for this device type (CSONHCA-1 for device A, CSONHCB-1 for device B, and so on).
BXACYL	This subfield of BXACDT contains the maximum addressable cylinder number for this device type. Initialization: This field is set to one less than the equate that defines the number of cylinders for this device type (CSONCMA-1 for device A, CSONCMB-1 for device B, and so on).
BXACINC	This subfield of BXACDT contains the cylinder increment factor. This number defines the interval at which the keypoint will be written to file. If, for example, the number is 25, the keypoint is written to file every 25 cylinders. On a system IPL, the exception recording pointers will be updated by this amount to ensure that the exception recording record sequence is valid. Initialization: This field should be set to provide frequent checkpointing without excessive filing of the capture and restore keypoint. A value greater than 255 should not be used because some segments access this field as 1 byte at BXACINC+1. SIP input = (KPUPA–KPUPD) DDCCAP.
BXADCD	This subfield of BXACDT contains the cylinder displacement to the duplicate records for this device type. This field is used by capture to calculate the location of duplicated records if the primary copies are not accessible and by restore to calculate the duplicate location if the duplicated records will be restored. Initialization: The system value for this device type will be inserted at system IPL time.
BXADHD	This subfield of BXACDT contains the head displacement to the duplicate records for this device type. This field is used by capture to calculate the location of duplicated records if the primary copies are not accessible and by restore to calculate the duplicate location if the duplicated records will be restored. Initialization: The system value for this device type will be inserted at system IPL time.
BXINTS	This tag refers to the module interval definition table contained in each CDT entry. The table defines to capture and restore the valid system modules that can participate in a capture or restore. It also defines whether duplicated records will be captured on the defined modules. Initialization: This tag has a length attribute equal to the total number of bytes required for the module interval definition table. The BXINTS field must be the same length in all CDT entries.

Table 9. Device Type Control Table (CDT) (continued)

Name	**Description**
BXDINT	This tag defines an entry in the module interval definition table. Each entry defines a range of valid modules for this device type and an indication of whether duplicated records will be captured. The entries do not need to be in any sequence and spares can exist but must be set to all hex zeros. Initialization: This tag has a length attribute equal to the size of 1 entry in the module interval definition table.
BXILOM	This subfield of BXDINT defines the low module in a range of valid modules to be captured or restored. Initialization: The hexadecimal module number for the low module in a module number range should be set here. SIP input = (INT*x*P or INT*x*N) DDCCAP.
BXIHIM	This subfield of BXDINT defines the high module number in a range of valid modules to be captured or restored. Initialization: This field is initialized to the high hexadecimal module number for a module number range. BXIHIM must be greater than BXILOM for a given range specification. SIP input = (INT*x*P or INT*x*N) DDCCAP.
BXICAT	This field contains an indication of whether the duplicated records on the modules in the range between BXILOM and BXIHIM will be captured. Initialization: Set to "P" if both nonduplicated and primary copies of duplicated records will be captured. Set to "N" if only nonduplicated records will be captured. SIP input = (INT*x*P or INT*x*N) DDCCAP, where *x* = A, B, C, or D.
BXALRT	This field is used by the restore function to save the lower limit on a THRU restore. Cylinder, head, and record are saved as CCHHR. The last byte of this field (record number) has the tag name BXALRR.
BXAHRL	This field is used by the restore function to save the upper limit for a THRU restore. Cylinder, head, and record number are saved as CCHHR. The last byte of this field (record number) has a tag name of BXAHRR.

Keypoint Equates

Keypoint equates define varieties of capture and restore keypoint information used during processing. Time intervals, delimiters, and indicators are maintained for hardware and capture and restore program control.

Table 10. Keypoint Equates

Name	**Description**
BXXINT	This equate defines the time interval, in minutes, in which the log time-stamp program will be started. This program (BXTT) will write a TME label record to all logging tapes each time it is activated. Initialization: SIP input = (STAMP) LOGCAP.

Table 10. Keypoint Equates (continued)

Name	Description
BXAIT3	This equate defines the time interval, in seconds, in which the capture and restore security program (BXAI) is to be started. This program determines if capture or restore read/write entries are processing longer than expected. If so, the operator is notified and if the condition continues, the capture or restore is paused. Initialization: It seems reasonable to set this value to 1/2 the amount of time required to perform the fastest module capture for the given system. SIP input = (SECUR) RESCAP.
BXADT1	This field is equated to BXAIT3.
BXABT1	This field is equated to BXAIT3.
BXAET1	This equate defines the maximum amount of time that will be allowed for individual entries to end after an ABORT ALL message is entered. If the entries do not end in the specified time, the operator will be informed. Initialization: SIP input = (ABRTO) RESCAP.
BXAMT1	This equate defines the number of times an internal tape mount will be attempted before ending the module. Initialization: SIP input = (TPERR, subparameter a) RESCAP.
BXAMT2	This equate defines, in seconds, the time interval between the successive tape mounts that will be processed. This equate is used with BXAMT1 to control the retry of an internal tape mount operation. Initialization: The value specified should allow the operator to respond to the problem that interfered with the previous tape mount attempt. SIP input = (TPERR, subparameter b) RESCAP.
BXANNRA	This equate defines the number of times capture will retry an individual disk error (independent of the error retry procedures). Initialization: SIP input = (DKERR, subparameter a) RESCAP.
BXDELIM	This equate defines the delimiter (/) that separates multiple modules or multiple tapes in a capture or restore input message.
BXDEL1	This equate defines the delimiter (blank) that separates modules from tapes specified in a selective start message for capture or restore.
BXDEL2	This equate defines the delimiter (-) that separates tape hardware addresses in a dual tape entry.
BX1MAX	This equate defines the maximum number of read/write entries that are allowed active simultaneously for capture or restore. Initialization: This field is set to 32.
LABEL	This equate defines the length (256) of a label record written to a KPT or XCP/LOG capture tape.
SMALLB	This equate defines the indicator that is used to denote small core blocks (X'21').
LARGEB	This equate defines the indicator that is used to denote large core blocks (X'31').
FORKB	This equate defines the indicator that is used to denote 4KB core blocks (X'51').

Table 10. Keypoint Equates (continued)

Name	Description
REELNO	This equate defines the number to be used to define the first reel of tape of a module capture.
TPCTL	This equate is used by the restore function as a control for the number of attempts to initially read the header of a tape to be restored. Initialization: This equate should be initialized to a logical value for error retries: the suggested value for retry is 3. SIP input = (TPERR, subparameter c) RESCAP.
BXAUMAX	This equate defines the number of times restore will retry an individual disk error (independent of the error retry procedure). Initialization: SIP input = (DKERR, subparameter b) RESCAP.
DEVA	This equate defines the device type indicator for device A (X'01').
DEVB	This equate defines the device type indicator for device B (X'02').
DEVC	This equate defines the device type indicator for device C (X'03').
DEVD	This equate defines the device type indicator for device D (X'04').
REALSC	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the starting cylinder of the interval that is being analyzed.
REALSH	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the starting head of the interval that is being analyzed.
REALSR	This equate is used by restore analysis as a tag name for that byte of the ECB work area that is used to save the starting record number of the interval that is being analyzed.
REALEC	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the ending cylinder of the interval that is being analyzed.
REALEH	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the ending head of the interval that is being analyzed.
REALER	This equate is used by restore analysis as a tag name for the byte of the ECB work area that is used to save the ending record number of the interval that is being analyzed.
ALTRSC	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the starting cylinder of the duplicated area for the interval that is being analyzed.
ALTRSH	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the starting head of the duplicated area for the interval that is being analyzed.
ALTRSR	This equate is used by restore analysis as a tag name for the byte of the work area that is used as a save area for the starting record number of the duplicated area for the interval that is being analyzed.
ALTREC	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the ending cylinder of the duplicated area for the interval that is being analyzed.
ALTREH	This equate is used by restore analysis as a tag name for the 2 bytes of the ECB work area that are used to save the ending head of the duplicated area for the interval that is being analyzed.

Table 10. Keypoint Equates (continued)

Name	Description
ALTRER	This equate is used by restore analysis as a tag name for the byte of the ECB work area that is used to save the ending record number of the duplicated area for the interval that is being analyzed.
BXSECT	This tag defines the BXSECT DSECT used by capture and restore.
Note: Fields BXCAP5 through BXAUVER are used by the capture and restore error recovery segments BXAN, BXBN, and BXAU.	
BXCAP5	This tag refers to the work area defined by the BXSECT DSECT.
BXSAVE	(RESERVED)
BXBXAH	(RESERVED)
BXBXAN	This tag refers to a work area that is used by the capture error recovery program, BXAN, and by restore error recovery program BXAU.
BXANRL	This field contains the record length of a record that was unsuccessfully trying to be read through a find-type macro.
BXANNOR	This field contains the number of records per track used by capture error recovery programs BXAN and BXBN.
BXANLCW	This field contains the last CCW error compare address used by capture error recovery programs BXAN and BXBN.
BXANFCW	This field contains the first CCW error compare address used by capture error recovery programs BXAN and BXBN.
BXANSW1	This field indicates whether the primary or duplicate track had an irrecoverable error.
BXANSW2	(RESERVED)
BXANRL	This field contains a count of the number of bytes remaining in the record being processed by capture error recovery segment BXBN.
BXANIOB	This field contains the base address of the record being processed by capture error recovery segment BXBN.
BXANLER	This field contains the last CCW error compare address that was processed by capture error recovery.
BXANNR	This field contains a count of the number of retries that have been done so far. It is used by capture error recovery.
BXANDM	This field contains the duplicate module number of the module number that caused a find error.
BXANS1	This field refers to the duplicate argument parameters (cylinder and head) used by the error recovery segments.
BXANDC	This field contains the duplicate cylinder that corresponds to the prime cylinder associated with a find error.
BXANDH	This field contains the duplicate head that corresponds to the prime head associated with a find error.
BXANRD	This field contains the record number of the last record read when a find error occurred.
BXANMR	This field contains the maximum record number possible on the track being processed by capture error recovery segment BXBN.
BXANCWS	This field is the read CCW save area used by error recovery segment BXBN.
BXAUFCW	This field contains the first CCW error compare address used by restore error recovery program BXAU.

Table 10. Keypoint Equates (continued)

Name	Description
BXAULCW	This field contains the last CCW error compare address used by restore error recovery program BXAU.
BXAUPCW	This field contains the address of the previous prime CCW group that caused an error. Used by BXAU.
BXAUDCW	This field contains the address of the previous duplicate CCW group that caused an error. Used by BXAU.
BXAUFIL	This field is a save area for the FDCTC macro used in restore error recovery segment BXAU.
BXAUREG	This field is the register save area for segment BXAU.
BXCLTIN	This field contains the base of the tape status table used by segment BXEL.
BXAUERR	This field contains a count of the number of retries attempted on prime tracks used in restore error recovery.
BXAUERD	This field contains a count of the number of retries attempted on duplicate tracks used in restore error recovery.
BXAUVER	This field is used as a switch to indicate whether the primary or duplicate track is currently being processed by segment BXAU.
BXSEEK	This field refers to the seek address of the primary track.
BXSEKB	This field is used to process the start of the seek address for the primary track.
BXSEKCA	This field contains the cylinder number of the prime module (used for CCW seek).
BXSEKH	This field contains the head number of the prime module (used for CCW seek).
BXSEKR	This field contains the record number of the prime module (used for CCW seek).
BXSK1	This field refers to the seek address of the duplicate track.
BXSK1B	This field is used to process the start of the seek address for the duplicate track.
BXSK1CA	This field contains the cylinder number of the duplicate module (used for CCW seek).
BXSK1H	This field contains the head number of the duplicate module (used for CCW seek).
BXSK1R	This field contains the record number of the duplicate module (used for CCW seek).
BXAUEND	This equate marks the end of the BXAU work area.
BXTPALL	This field contains a tape header for use by capture and restore tape processing.
BXTPLBL	This field contains the tape label (BXA).
BXTPCCW	This field contains the current buffer size index used in building CCWs.
BXTPDEV	This field contains the device type indicator for the current module that is being processed.
BXTPDTE	This field contains the capture date.
BXTPTME	This field contains the capture time.
BXTPSEQ	This field contains the tape block sequence number that is currently being processed.

Table 10. Keypoint Equates (continued)

Name	Description
BXXMODA	This field contains the module number of the current module that is being captured or restored.
BXXRNO	This field contains the reel number of the current reel that is being processed.
BXTPSS	(RESERVED)
BXTPSPA	(RESERVED)
BXTPCNT	This field refers to the capture and restore count fields.
BXTPCTF	This field contains the cylinder/head address of each record that is being processed.
BXTPSP1	(RESERVED)
BXTPCF2	(RESERVED)
BXTPAL2	This field contains the duplicate tape header area.
BXTPCT2	This field contains the capture and restore duplicate count fields.
BXCBADR	This field defines the area containing the address of a core block used for I/O buffers.
BXCBADF	This field contains the address of a core block that will be used for I/O buffers.
CAPCCW	This field defines a work area used by capture and restore containing disk and tape CCW areas.
CAPREG	This field is used as a save area for various capture and restore fields.
CAPDSK	This field contains the capture and restore disk CCW build area.
CAPTAP	This field contains the capture and restore tape CCW build area.
CAPRTR	This field contains the retry seek address.
RESDK2	(RESERVED)
RESCW2	(RESERVED)
SAFETY	This field redefines the first 24 EBW bytes of the ECB and contains information common to capture and restore.
SAFIPD	This field defines an area containing various capture and restore indicators.
SAFOPI	This field indicates whether a capture or restore is active.
SAFSRT	This field indicates whether a module is completed or not.
SAFIPFD	This field contains the address of the appropriate in-progress field (IPF).
SAFMOD	This field contains the current module number being processed.
SAFDE2	This field contains the current tape hardware address.
SAFDES	This field contains the symbolic tape name.
SAFTMM	This field contains the decimal time that the capture or restore entry began.
SAFTMB	This field contains the binary time that the capture or restore entry began processing the current reel of tape.
SAFPSS	This field contains the capture common area address (CMMPSS).

Keypoint DSECTs

The following DSECT definitions are provided in the capture and restore keypoint record (BXAXF).

Disk Device Control Table (DDCT)

This record is completely initialized by the online capture and restore programs as needed.

Table 11. Disk Device Control Table (DDCT)

Name	Description
DDCT	DSECT describing the disk device control table.
DDCLAB	This field contains the program record number and demand counter that are used in lieu of an alphabetic record ID because the disk device control table is maintained as a program.
DDCNAM	This field contains the program stamp. This field will always contain BXAM because this is the only segment that files the DDCT.
DDCSYN	This field contains a synchronization field used to ensure that the capture and restore keypoint and the DDCT reflect the same point in time when a system IPL occurs. Refer to keypoint tag BXSYNF for more information.
DDCTOT	This field contains the number of modules to process.
DDCDON	This field contains the number of modules that are currently done.
DDCCIP	This field contains the number of entries in progress.
DDCCES	This field contains the total number of restore entries started.
DDCPIND	This field contains the processor participation indicator.
DDCTMID	This field contains the number of modules that are deferred.
DDCTMOF	This field contains the number of modules that are offline.
DDCHDR	This tag refers to the first 30 bytes of the DDCT, which are considered the record header.
DDCNMOD	This tag is equated to the number of DDCMOD entries in the DDCT. It is calculated as follows: 4096 − L'DDCHDR − 1.
DDCMOD	This byte field is used as 8 1-bit indicators to maintain the status of modules that are being captured or restored. 4065 of these fields exist in the DDCT, allowing for a module number range of 0–4064.

Database Reorganization

This chapter describes the functions that are available in database reorganization (DBR) and how to use these functions. You should have a general knowledge of the TPF system; see *TPF Concepts and Structures* and *TPF System Generation*.

DBR allows you to:

- Capture the fixed file database and reload it on another system with a different or reorganized database.
- Capture the pool database and reload it on another system with a different or reorganized database.
- Capture only specified record types and reload them as necessary.
- Bypass record types or certain records in a record type while capturing the database.
- Control the processing of the utility while it is running.
- Run in any system state while capturing the database.

DBR also allows a subsystem user to capture an existing database (output phase) and reload the captured database on a new subsystem (input phase).

DBR is started by the ZDBRO and ZDBRI commands. In a multiple database environment, the ZDBSO and ZDBSI commands are also available, which allow you to process more than one subsystem user in a single message. See *TPF Main Supervisor Reference* for additional information about the multiple database function (MDBF). See *TPF Operations* for information about the format of the DBR commands.

Prerequisites

The DBR process requires:

- A fixed record type of #DBRRI allocated with the number of record types (fixed and pool) in the system plus 3 records. These records are used to control DBR processing and define which records to include in the capture.
- A DBF tape if you want to capture any fixed file records.
- A DBP tape if you want to capture any pool records.

Considerations Before Using DBR

You should have a complete knowledge of the database to be captured and be able to:

- Define the system state the DBR run will operate in. Table 12 summarizes the operational requirements for starting a DBR run.
- Define which record types can or cannot be captured.
- Define which pool records will be captured.

Table 12. Database Reorganization Operational Requirements

	Phase	**Record Type**	**IPL**	**State**	**Other**
1	Input	Fixed	General file	1052	None
2	Input	Pool	Prime module	1052	None
3	Output	Fixed	Prime module	1052	None

Table 12. Database Reorganization Operational Requirements (continued)

	Phase	Record Type	IPL	State	Other
4	Output	Fixed	Prime module	NORM	Logging
5	Output	Pool	Prime module	1052	None
6	Output	Pool	Prime Module	NORM	Logging

For example:

- Line 3 shows that if you want to start a DBR run for the output phase of fixed records, you must IPL the prime module and the TPF system must be in 1052 state.
- Lines 4 and 6 show that the system can enter NORM state only if record logging for the capture and restore utility is active. See "Capture and Restore" on page 55 for more information about capture and restore.
- Line 1 shows that if you want to start a DBR run for the input phase of fixed records, you must IPL the general file and the TPF system must be in 1052 state.

These operational requirements apply only to the START and RESTART functions; that is, the actual running of DBR. The other DBR commands are not subject to these requirements.

Additional considerations include the following:

- If a processor is running DBR, you cannot start another DBR run on the same processor.
- In a multiple database function (MDBF) system, whether loosely coupled or not, you cannot start a DBR run for a subsystem that is currently running DBR.
- In an MDBF system, whether loosely coupled or not, you cannot start a DBR run for a subsystem user that is currently running DBR.
- In a loosely coupled MDBF system, 1 processor can run DBR for a given subsystem user while another processor runs DBR for a different subsystem user.
- In a loosely coupled MDBF system, 1 processor can run DBR for a given subsystem while another processor runs DBR for a different subsystem.
- DBR cannot be run on tapes mounted in blocked mode.

  If you want to run the DBR output phase with buffered tape devices, you must do one of the following:

  - Specify the DBRBUF=YES parameter on the CONFIG macro in your SIP stage 1 deck. See *TPF System Generation* for more information about the CONFIG macro.
  - Enter the ZSYSG command with the DBRBUF parameter.

  Attention: Running the DBR output phase with buffered tape devices will improve performance; however, if the system re-IPLs or tape errors occur while DBR is in progress, you may lose data. Also, because the switch that is set by the DBRBUF parameter is subsystem-shared, you can specify ZSYSG with the DBRBUF parameter only from the basic subsystem (BSS).
- DBR uses 2-byte hexadecimal record types and 8-byte ordinal numbers (up to 16 digits).

Note: All input and output messages will handle ordinal numbers as hexadecimal values rather than decimal values.

Database Reorganization Control Records

This section describes the control records used with DBR. These control records are defined in the DB0DB data macro.

Master Keypoint

The master keypoint contains information necessary to control the capture of the fixed file and pool database. It contains information relating to each fixed file record type and pool record type available in the system. The master keypoint is a 4KB record that contains bits for all possible record types in the system. These bits indicate:
- If the record type will be captured.
- If the record type contains records that will be bypassed.

In addition, the master keypoint contains the available ECB count. The default ECB count is 4; however, you can modify this.

Working Keypoint

The working keypoint is used to control the processing of DBR during output and input phases. When you enter a START command, the master keypoint is copied to the working keypoint to prevent the loss of the master keypoint if a system error or restart occurs. The DBR code periodically saves the last record type processed in the DBR working keypoint so that you can use the RESTART function to restart DBR at that record type.

Override Keypoint

The override keypoint is used to process a START *rectype* END command. This allows you to capture selected record types or groups of records in a record type without disturbing the master keypoint. Each record type that you enter turns on a bit in a 4KB record associated with the override keypoint to show that record type is to be captured. Once the END parameter is found, the override keypoint is copied to the working keypoint and the DBR output phase is started to capture the selected record types.

Database Reorganization Exception Records

The exception records control which records in individual record types will be bypassed during the DBR output phase. Each ordinal number range of a bypass command for a record type is placed in this record. The exception record can contain up to 256 different sets of ordinal numbers. When the output phase is running, each record type is checked to see if exceptions exist. If exceptions exist, each ordinal number is compared to the exception entries to see if it should be bypassed.

Database Reorganization Processing Description

Database reorganization (DBR) consists of 3 phases:

Phase	Description
Initialization phase	Sets up the control and exception records to reflect the records to be captured.

Output phase	Captures the specified records and writes them to tape.
Input phase	Reloads the captured records to a new TPF system.

Initialization

Before starting DBR, you must initialize the control and exception records that are used by DBR. The following commands are used in the initialization phase:
- ZDBRO INIT or ZDBSO INIT
- ZDBRO BYPASS or ZDBSO BYPASS
- ZDBRO RESET or ZDBSO RESET
- ZDBRO DISPLAY or ZDBSO DISPLAY

You can also use the ZDBRO OECB or ZDBSO OECB command to modify the number of available ECBs to fit your installation requirements. The default number of available ECBs for the output phase is 4. Be careful when raising the available ECB count because each ECB causes as many as 20 frames to be used. Checks have been inserted in the system to prevent users from running out of frames during the running of DBR. However, additional activity on the system and the running of DBR on multiple subsystems simultaneously can cause working storage to be used up rapidly.

INIT

You must enter the ZDBRO INIT or ZDBSO INIT command before any other DBR command.

The INIT function formats the predefined record type (#DBRRI) to reflect all the fixed file and pool records to be captured into the necessary control and exception records that are needed by DBR.

- The master keypoint is formatted to reflect all in-use fixed file and pool records that are to be captured. There is a maximum of 12 288 possible fixed file record types. In addition, the available ECB count is set to 4 (default).
- The default initialization sets up the control record to capture all the records that are accessible from the issuing SSU/processor/I-stream triplet. If an SSU/processor/I-stream *triplet* is specified as a parameter, the control record is set up to capture all the records that are owned by the specified SSU/processor/I-stream.
- The fixed file and pool exception records are formatted to zero with an available count of 256 entries.
- The ZDBSO message is used to capture all of the records that a particular subsystem user owns. For example, if you want to capture all the records owned by SSU3, SSU4, and SSU5, enter:

  ZDBSO INIT SSU3/SSU4/SSU5
- The ZDBRO *triplet* command is used to capture all records that the SSU/processor/I-stream combination owns. Without the *triplet* options, ZDBRO is used to capture all records that the SSU has access to; the SSU may or may not own these records. For example, if you want to capture all the records owned by SSU1 on processor B I-stream 2, enter:

  ZDBRO INIT SSU1/B/2
- Only 1 ZDBxO message can be entered for a single SSU at a time. If another ZDBxO message is required for additional database capturing, you need to wait for the DBR output phase to finish completely before entering another ZDBxO message.

Once the records are initialized, you can modify the control records as necessary to process the database as determined by your installation.

BYPASS

Use the BYPASS function to set a certain record type or group of records in a record type so they are not captured. For example:

- If you do not want to capture hexadecimal record type 0009, enter:

  ZDBRO BYPASS RECtype-0009

  or

  ZDBSO BYPASS RECtype-0009

  The BYPASS option sets the control bits that are ignored or preserved by the BYPASS parameter of the INIT command. When BYPASS=YES on the INIT function, the capture bits that are set with the BYPASS function are ignored.

  A practical instance of the BYPASS function involves the PROT table. The output tape device used by DBR is identified in the PROT table. This table is captured by the DBR output phase and restored by the DBR input phase. When the restored system comes online, the previous PROT table takes effect and may not agree with the tape status of the current system unless the same tape device was specified for both phases of DBR. If you cannot avoid using different tape devices, you can bypass the capture of PROT during the output phase, or correct the contents of PROT after the system is restored if the BYPASS function was not used.
- If you want to capture hexadecimal record type 0009, but do not want to capture ordinals 020-035 in that record type, enter:

  ZDBRO BYPASS RECtype-0009 RANGE-020.035

  or

  ZDBSO BYPASS RECtype-0009 RANGE-020.035
- If you want to capture only pool records that are in use, enter:

  ZDBRO BYPASS DIR

  or

  ZDBSO BYPASS DIR

RESET

If a particular hexadecimal record type or group of records in a record type are set so that they will not be captured, but you determine that they should in fact be captured, use the RESET function to reset the capture bit in the master keypoint and the exception fields in the exception records.

For example:

- If you bypassed hexadecimal record type 0009, when it should be captured, enter:

  ZDBRO RESET RECtype-0009

  or

  ZDBSO RESET RECtype-0009
- If you bypassed ordinal numbers 020-035 in hexadecimal record type 0009, and now want to capture them, enter:

  ZDBRO RESET RECtype-0009 RANGE-020.035

or

ZDBSO RESET RECtype-0009 RANGE-020.035

- If only pool records that are in use are set to be captured, but you want to capture all pool records, enter:

  ZDBRO RESET DIR

  or

  ZDBSO RESET DIR
- If you entered a START *rectype* command but did not enter START *rectype* END and want to reset the START *rectype* parameters, enter:

  ZDBRO RESET OVR

  or

  ZDBSO RESET OVR

DISPLAY

Use the DISPLAY function to display the status of the DBR control and exception records to determine if the records that are set to be captured or bypassed are correct, or changes are necessary to update these records before starting the output phase.

Note: Do not use the display message after DBR is started. The results of the display are not valid in this case.

For example:

- If you want to display the hexadecimal record types that are set to be captured, enter:

  ZDBRO DISPLAY ALL RECtype-*value*

  or

  ZDBSO DISPLAY ALL RECtype-*value*

  where *value* is the record type code, which must be a 4-digit hexadecimal number. The record type that is specified by *value* is the first record type to be displayed. A maximum of 48 items can be displayed in each output message.
- If you want to display the hexadecimal record types that are set on in the override keypoint, enter:

  ZDBRO DISPLAY OVR

  or

  ZDBSO DISPLAY OVR
- If you want to display the hexadecimal record types that have exception entries, enter:

  ZDBRO DISPLAY EXC

  or

  ZDBSO DISPLAY EXC
- If you want to display the exception entries for a given hexadecimal record type, enter:

  ZDBRO DISPLAY RECtype-*value*

  or

ZDBSO DISPLAY RECtype-*value*

where *value* is the record type code, which must be a 4-digit hexadecimal number.

Output Phase

When the DBR control and exception records are initialized, you can start the DBR output phase.

Start the DBR output phase with the ZDBRO START or RESTART command or with the ZDBSO START or RESTART command. See *TPF Operations* for additional information about the format of all the DBR commands.

Before you start the DBR output phase, display the master keypoint and exception records to ensure the database will be captured correctly.

As each record type is captured during the output phase, a message is sent to the console. Check the record type to ensure that the correct record types are being captured.

At any time during the output phase, you can request status with the STATUS function. The record type, ordinal number, and number of available ECBs are displayed on the console. The STATUS function is not available during the input phase.

If the system needs to be IPLed for any reason while DBR is running, DBR forces the DBF/DBP tapes to be removed before DBR is started again.

- The command handler checks to ensure that the record initialization step is complete and DBR is not already running. If an error is found, a message is displayed and the command is ignored.
- The working keypoint is formatted to control the record capture unless restart was entered which causes the file copy of the working keypoint to be obtained to find the last record type processed.
- The controller searches (starting with hexadecimal record type 0000, for START and the last record type processed for RESTART) the capture bits until one is found that is in use.
- Each ordinal number (starting with hexadecimal 00) for the record type is compared against the exception entries (if present) and passed through the FACE table for a valid file address.
- If the record is acceptable, the controller moves the ordinal number and file address of the record to a 4096-byte header record. The header record contains the record type that is being processed, the number of records in the tape block, the size of the records, and the ordinal number and file addresses of each of the records in the subsequent tape block. The tape block can contain as many as 160 small records, 48 large records, or 16 4KB records.
- The controller then goes to the next ordinal number in the record type and processes it as described previously. When there are enough records to fill a tape block or when the end of a record type has been reached, a CREEC macro is issued to the disk read/tape write routine.
- As each CREEC macro is issued to the disk read/tape write routine, a count of outstanding CREECs increases. If the count is greater than the number of available ECBs set in the keypoint, the controller defers to allow the outstanding

CREECs to complete processing. Once the count reaches the available ECB count, processing continues. If the count is greater for an extended time frame, DBR ends.

- The DBR code will also wait in a loop until enough frames are available to process the records. When enough frames are available, the CREEC macro is issued.
- The CREEC macro passes the header block that was created by the controller to the disk read/tape write routine.
- The disk read/tape write routine finds all of the records whose ordinal numbers and file addresses have been passed in the header block. It then writes the header block and all of the records that it finds to the output tape.

  For maximum efficiency, as many as 10 FINDC macros are issued before a WAITC macro is issued. If a find error occurs, a core block is zeroed and the record ID field is set to X'FFFF'. This is a dummy record containing all zeros, and is referenced in the online message as the ZZ record. If the record is necessary, you must correct the record and restart DBR, or correct the record after it is moved to the new database.
- If a tape error occurs, the tape is closed and a message requesting a new tape is sent to the console. Once the new tape is mounted, you must enter the SWITCH command to allow DBR to open and start using the new tape.
- While the controller is processing, it periodically checks the abort control bit in the keypoint. If you enter an ABORT command, the controller closes the tape, cleans up, and exits. (See “Stopping DBR” on page 97 for more information about the ABORT function.)
- Once all fixed record types are processed, the controller closes the fixed tape, turns on the pool control bit, and goes to the pool controller to process the pool records.
- The pool controller processes the pool records based on the pool capture bits pointed to by the keypoint in the same way as fixed. Upon completion of the pool record types, the pool tape is closed and the output phase ends.

Unless the DBR control records are destroyed, the DBR output phase can be stopped and restarted at any time. The restart starts processing with ordinal number zero of the record type that was being captured when the process was stopped. The DBF/DBP tape that was mounted when the process was stopped should be saved and a new tape mounted when the restart is requested.

Records not Captured

When capturing records, the records associated with TPF programs, loaders, and images are bypassed. Records that are bypassed include:

- Core image restart area (CIMR) records
- Keypoint records
- IPL area records
- Keypoint staging area records
- E-type program area records
- Program history area records
- Image pointer records (CTKX)
- Any record types that have RESTORE=NO coded on the RAMFIL macro, including #XPRG records containing (ISO-C) C programs.

If you want to move these records from 1 TPF system to another you can do a full load or use the ZIMAG command to create or copy the image.

Tape Mount in MDBF Environment

The tape utility writes the subsystem user ID to a tape label. This subsystem user ID is checked when the tape is mounted. Remember, DBR cannot be run on tapes that are mounted in blocked mode.

Logging

If you run DBR in a state other than 1052 state, record logging for the capture and restore utility must be started to record any changes that are made to records after DBR has captured them to tape. Logging must remain active until the system is cycled down to 1052 state. See “Capture and Restore” on page 55 for additional information about the capture and restore utility.

Stopping DBR

If you must stop the DBR process during the output phase, use the ZDBRO ABORT or ZDBSO ABORT command. This command closes any open tapes and resets the DBR control bits. The DBR code periodically saves the last record type processed in the DBR working keypoint so that you can restart DBR at that record type after an unplanned shutdown of the system. You can use the RESTART function to restart DBR at the record type saved in the keypoint. Because the record types that were written to tape may not be in sequential order, additional record types after the specified record type also may have been written to tape. The DBR code ensures that all record types up to the specified record type were processed before it saves that record type in the keypoint. If an ABORT command was entered, a RESTART command restarts DBR at the last record type processed before the ABORT message was entered. DBR restarts with ordinal number 0 of the record type that was being processed when the system was lost. You can also use the RESTART function to restart DBR at any record type in order to avoid writing records to tape that may have already been written before an unplanned system shutdown.

Input Phase

The DBR input phase is started by the ZDBRI START or RESTART, or the ZDBSI START or RESTART command. See *TPF Operations* for more information about the format of all the DBR commands.

- The command handler checks to ensure that DBR is not currently running and the system is in 1052 state. Fixed files are loaded from the general file and pool records are loaded from the online system.
- The working keypoint is created or read (if already created) and the restart record type is posted if restart was entered or record type zero for a normal start.
- The controller opens the appropriate input tape and reads the first header record from the tape.
- The controller increments the CREEC counter and checks it against the available ECB count in the keypoint. If all available ECBs are used, the controller defers waiting for some disk write entries to be completed.
- The DBR code also waits in a loop until enough frames are available to process the records. When enough frames are available, the CREEC macro is issued. You can modify the available ECB count to allow more or fewer CREEC macros to be issued (default is 4). Be careful when raising the available ECB count because each ECB causes as many as 20 frames to be used. Checks have been inserted in the system to prevent users from running out of frames during the running of DBR. However, additional activity on the system and the running of DBR on multiple subsystems simultaneously can cause rapid working storage depletion.
- The CREEC macro passes the header block read in by the controller to the disk write routine.

- The disk write routine reads the next tape block from the input tape and files the records contained in the tape block to disk by taking each record’s file address and moving it to a file address reference word in a data event control block (DECB) before filing it.

  For maximum efficiency, as many as 10 FILNC macros are issued before a WAITC macro is issued.
- The controller reads the next header record from the input tape when the disk write routine has turned off a bit in the DBR keypoint, indicating that it has read in a tape block.
- If permanent errors occur on the tapes during the input phase, the tape is closed and the program exits. You must remake or correct the tape and restart the input phase.
- The controller periodically checks a bit in the keypoint to see if the disk write routine is done with the input tape. If yes, the tape is closed and the program exits.
- If the record cannot be filed, a message is sent to the console, the CREEC count decremented, and the program exits.
- If the record is a dummy error record from the output phase (record ID = X'FFFF'), a message is sent to the console before processing the record. The record should be corrected after the input phase ends.
- At the end of file, the tape is closed and the program exits.

The DBR input phase can be restarted at any specified record type. The tape is searched until the specified record type is found and processing continues from that point to end of file.

Logging

Once the fixed file and pool database are reloaded, you should process the logging tapes that were created during the output phase (if in other than 1052 state). When the logging tape restore has been completed, the database reorganization process is completed.

Database Reorganization Sample Problem

The following shows an example of how a large fixed record type (0000) is captured from one TPF system and reloaded to a different TPF system. It explains the general flow of the DBR utility and the sequence to follow while using the utility.

Before you begin: Start DBR output in 1052 state and IPL the prime module to 1052 state.

Note: If the system is in a state other than 1052 state, capture and restore logging must be active before you start DBR.

1. Enter **ZDBRO INIT** to initialize the DBR record:

   ```
   User:   ZDBRO INIT

   System: DBRO0000I 13.42.33 REQUEST COMPLETE
   ```

   Note: DBR initialization must be the first step of the DBR process.
2. Enter **ZDBRO DISPLAY ALL** to display the hexadecimal record types to be captured:

```
User:   ZDBRO DISPLAY ALL

System: DBRO0067I 10.22.21 RECORD TYPES TO BE CAPTURED
        0000 0001 0002 0003 0004 0005 0006 0007
        0008 0009 000A 000B 000C 000D 000E 000F
        0010 0011 0012 0013 0014 0015 0016 0017
        0018 0019 001A 001B 001C 001D 001E 001F
        0020 0021 0022 0023 0024 0025 0026 0027
        0028 0029 002A 002B 002C 002D 002E 002F
        MORE
```

Note: The first 48 in-use hexadecimal record types are displayed. MORE, which is on the last line of the display, shows that there are more in-use record types.

3. Enter **ZDBRO BYPASS** as follows to bypass ordinal numbers 110–119, 280–287, and 17–28:

```
User:   ZDBRO BYPASS REC-0000 RAN-6E.77

System: DBRO0000I 12.54.00 REQUEST COMPLETE

User:   ZDBRO BYPASS REC-0000 RAN-118.11F

System: DBRO0000I 12.55.00 REQUEST COMPLETE

User:   ZDBRO BYPASS REC-0000 RAN-11.1C

System: DBRO0000I 12.56.00 REQUEST COMPLETE
```

Note: The previous ordinal numbers are not needed and will be bypassed in this example.

4. Enter **ZDBRO RESET REC-0000 RANGE-11.1C** to reset ordinal numbers 17–28:

```
User:   ZDBRO RESET REC-0000 RAN-11.1C

System: DBRO0000I 12.58.00 REQUEST COMPLETE
```

Note: The previous ordinal numbers should be captured; they are reset.

5. Enter **ZDBRO DISPLAY REC-0000** to display hexadecimal record type 0000:

```
User:   ZDBRO DISPLAY REC-0000

System: DBRO0077I 13.45.28 EXCEPTIONS FOR RECORD TYPE 0000
         START                    END
        000000000000006E       0000000000000077
        0000000000000082       0000000000000083
```

Note: This display shows verification of the exception entries for the record type.

6. Enter **ZDBRO START 0000 END** to start the DBR output phase:

```
User:   ZDBRO START 0000 END

System: DBRO0010I 13.55.01 DBR OUTPUT PHASE STARTED
        COSK0079A 13.55.01 *CP* - MOUNT DBF TAPE FOR OUTPUT

User:   ZTMNT DBF 281 AO BP

System: COTN0046I 13.55.01 TMNT - TAPE DBF MOUNTED ON DEVICE 281
```

Note: Only hexadecimal record type 0000 is needed. END was entered so that the DBR process starts to capture the record type. The DBF (fixed) tape is requested and a mount is issued.

7. Enter **ZDBRO OECB 20** to change the default available ECB count:

```
User:   ZDBRO OECB 20

System: DBRO0000I 14.33.21 REQUEST COMPLETE
```

Note: The default value is 4. Be careful when raising the available ECB count because each ECB causes as many as 20 frames to be used. Consider gradually increasing the ECB count and monitoring system performance.

8. Enter **ZDBRO STATUS** to request DBR status:

```
User:   ZDBRO STATUS

System: DBRO0062I 22.12.33 DBR IS PRESENTLY PROCESSING
        RECORD TYPE 0000,ORDINAL NUMBER 000000000000013D,ECB= 00000020
```

The end of the record type is displayed in the following example:

```
System: DBRO0015I 22.45.33 DBR HAS FINISHED PROCESSING RECORD TYPE 0000
```

The end of output phase is displayed in the following example:

```
System: DBRO0016I 22.46.33 DBRO FIXED OUTPUT PHASE COMPLETE
        DBRO0011I 22.46.33 DBR OUTPUT PHASE COMPLETE
        COTC0080A 22.46.33 TCLS - REMOVE DBF FROM DEVICE 281
```

Note: If pool records were to be captured, the pool controller would receive control at this point and request the DBP (pool) tape.

9. Enter the following to IPL the general file:

```
User:   I 3E7
```

Note: The fixed file input phase must be run from the general file in 1052 state. The IPL address (3E7) may differ.

10. Enter **ZDBRI START FIXED** to start the input phase:

```
User:   ZDBRI START FIXED

System: DBRI0020I 09.30.16 DBR INPUT PHASE STARTED
        COSK0079A 09.30.16 *CP* - MOUNT DBF TAPE FOR INPUT

User:   ZTMNT DBF 281 AI BP
System: COTN0046I 09.30.16 TMNT - TAPE DBF MOUNTED ON DEVICE 281
```

Note: DBR requests the DBF tape that was created in previous steps as input and a mount is issued.

11. Enter **ZDBRO IECB** to change the default available ECB count:

```
User:   ZDBRO IECB 20

System: DBRO0000I 09.31.43 REQUEST COMPLETE
```

Fixed hexadecimal record type 0000 is reloaded and the DBR process ends in the following example:

```
System: DBRI0015I 09.33.17 DBR HAS FINISHED PROCESSING RECORD TYPE 0000
        DBRI0000I 09.33.17 REQUEST COMPLETE
        COTC0087A 09.33.17 TCLS - REMOVE DBF FROM DEVICE 281 VSN 111111
```

See *TPF Operations* for more information about the ZDBRO commands.

Recoup

Recoup processing verifies fixed records and the chains of pool records attached to them. Recoup processing returns file pool addresses that were released by applications or were lost because of software errors or unplanned system restarts.

Note: When operating in a loosely coupled complex with more than eight processors, only the first eight processors can participate in recoup processing and all processors that participate must be at the same PUT level.

Fixed Records

Fixed records are predefined data records whose addresses can be calculated using the file address compute (FACE) program. These records are always present in the system and are usually those records that are accessed most frequently.

Pool Records

File pool records are records created on demand by applications programs. These records are always accessed indirectly by file address pointers in:
- Fixed records
- Main storage tables
- Other chained pool records.

Controlling the Use of Pool Addresses

The use of the file pool area is controlled by the control program file directory system. This system maintains directories for each of the pool areas (4KB, large, or small; duplicated or nonduplicated; and long-term or short-term records). Pool areas use file address reference format (FARF) addressing.

When an application program requires a file address for a created record, it issues a file address request macro to the control program. The control program searches the appropriate file pool directory for the first address that is not in use and, when it finds it, sets an indicator bit showing that the address is in use, and passes the file address to the application program.

When an application program determines that a file address is no longer required, it issues a file address release macro to the control program. For a short-term file pool record, for example, a record that exists only for the time of a transaction, the control program sets the appropriate indicator in the relevant directory to *not in use* state. For a long-term record, the control program writes the address of the released record to the real-time tape RTA and released pool address (FC33) records, which will be processed by directory support programs to update the status of the long-term file pool directories (see “Maintenance Functions” on page 23).

Losing Pool Addresses Because of Software Errors or a System Restart

Pool addresses can be lost when application programs fail to release file addresses correctly. Although the file copies of the file directory records are frequently updated from the main storage copies of directory records, at any point in time the file copies may not accurately reflect the actual state of the directories.The frequency with which the file copies are updated is based on the number of file addresses given out and placed in NOT AVAILABLE state. If an unplanned system restart occurs because of a machine malfunction or catastrophic error condition, the file pool directory records are read from file and additional record address NOT AVAILABLE indicators are set. The number that is set is an estimate of the maximum number of record file addresses that could have been set to NOT AVAILABLE state in the main storage directory since the file records were last updated. This eliminates the possibility of the same file address being given to application programs before and after a restart. However, some of the file addresses that are set to NOT AVAILABLE state may not have been used before the restart, so some file pool addresses are lost to the system.

Recouping Lost Pool Addresses

The recoup function reconciles the long-term file pool directories with the actual status of the pool area to provide summaries that can be used to isolate the causes of directory discrepancies. To do this, recoup accesses every fixed record and main storage table that references pool records. Recoup then reads any pool records referenced from those records. The data gathered during this record accessing and chain chasing phase can then be used to create file pool directories showing the actual status of the file pool area, and to produce summaries, which may be used to determine how errors are caused.

Recoup Functions by Phase

The following sections list the recoup functions by phase.

Recoup Pre-Phase 1 Functions

Phase 1 chain chasing is determined by information in the recoup descriptor container records and TPF collection support (TPFCS) recoup indexes. Create descriptor container records and recoup indexes describing all records and TPFCS collections that can reference long-term file pool addresses and TPFCS persistent identifiers (PIDs).

Recoup Descriptor Container Records

The descriptor container records are read and each record type in them is passed to the appropriate recoup chain chase program for processing. Several descriptor container records are provided with the recoup package. These records describe the processing to be done on record types which must be part of the TPF system and that can contain long-term file address and PID references. These descriptor container records can be used as an example when you are creating descriptor container records. See the GROUP and INDEX macros in *TPF System Macros* for additional examples and instructions.

Recoup currently supports the following types of record structure processing:

- Standard forward and backward chaining
- Nonstandard forward and backward chaining
- Records containing embedded addresses or PIDs found at a fixed or constant displacement in the record
- Records containing groups of addresses or PIDs either at fixed or variable locations.
- Records with multiple items containing fixed record ordinal numbers
- In a loosely coupled or multiple database function (MDBF) environment, records that are unique to a processor or subsystem user
- Records that make up a TPFCS collection that is contained in a data store

- Records in a TPFCS collection whose PID is contained as an embedded reference in another collection. The embedded reference is at a fixed displacement in a collection element
- Records in a file address structure that is contained as an embedded reference in a TPFCS collection. The embedded reference is at a fixed displacement in a collection element
- Record types that have a unique attribute (such as SSU, processor, or I-stream unique) will be chased by the record owner as defined in the FACE table (FCTB).

TPFCS Recoup Indexes

TPFCS collections created by user applications will not be recouped unless they are explicitly made known to the TPF system by establishing one or more TPFCS recoup indexes. A recoup index describes the location of persistent identifiers (PIDs) and file addresses embedded in all collections associated with that recoup index. A TPFCS database consists of automatically generated system collections as well as user collections. An anchor collection which is usually the application dictionary of the data store (named DS_USER_DICT), refers to user-created collections, which refer to other collections, and so on. Every user-created collection must be referred to or it will not be recouped. Several recoup indexes are provided with the recoup package to recoup the system collections. A recoup index must be created and associated with the anchor collection and any user collections that contain embedded references. This is done by running a user application or by entering the ZBROW RECOUP command. Furthermore, if the embedded reference is a file address, a TPF descriptor with the USE=TPFCS parameter on the GROUP macro must be created for that record.

If a TPF file contains an embedded PID, the TPF descriptor for that record is all that is needed to recoup the embedded PID. However, if the embedded PID contains additional embedded information, a recoup index will be needed to describe the embedded PID.

For more information about TPFCS recoup, see “TPFCS Recoup” on page 149. See the GROUP and INDEX macros in *TPF System Macros* for additional examples and instructions. See *TPF Operations* for more information about the ZBROW RECOUP command.

Recoup Considerations for TPF Internet Mail Server

Before you run recoup for the TPF Internet mail server database, ensure the mail server recoup descriptor (BKD1) is defined correctly and loaded on the TPF subsystem in which you plan to run the TPF Internet mail servers. You must also ensure that BKD1 contains a GROUP and INDEX macro pair associated with each #MAIL*xx* record that you have defined.

See *TPF Transmission Control Protocol/Internet Protocol* for more information about recoup considerations for the TPF Internet mail server database and how to update BKD1.

Setup

Recoup setup must be done once from CRAS state or above for each subsystem by entering the ZRECP SETUP command. The recoup setup process will create the recoup scheduling control table (IRSCT) and one recoup active root table (IRART) for each defined processor.

Recoup Phase 1 Functions

Recoup phase 1 processing consists of the following:

- Pre-chain chase processing
- Chain chase processing
- Post-chain chase processing.

Pre-Chain Chase Processing

Recoup phase 1 does the following before it begins to chase pool addresses:

- Switches released address (FC33) records and the RTA tape to distinguish between pool addresses that were released before recoup chain chase processing starts and pool addresses that were released during chain chase processing. Any pool addresses that are released by applications during chain chase processing are recorded on new FC33 records and a new RTA tape.
- Captures the long-term file pool directory records onto the phase 1 captured SONRI directory (#STPKP). This capture is done to provide a comparison for recording long-term file pool addresses that were given out between phase 1 and phase 3 of recoup.
- Initializes and verifies pseudo directories.
- Initializes ID tables.
- Builds the recoup scheduling control table (IRSCT) that determines how records are chain chased.
- Initializes the recoup active root tables (IRARTs) for each processor that is defined.
- Displays file pool counts.

Chain Chase Processing

Chain chase processing is controlled by the IRSCT. This table is built at the time the ZRECP RECALL command is entered, during a full recoup run. Each base group identified in the recoup descriptor container records (BKD) initialized during the pre-phase 1 stage is examined for validity and scheduling directives. These groups are added to the IRSCT along with all TPFCS data stores that reside on the current subsystem. Recoup uses this information to define the order, and in a loosely coupled complex; on which processor the chain chase will be run. With this information, every record on the subsystem that contains pointers to file pool records is accessed. Those file pool records and all file pool records subsequently chained from them are accessed until all pool-related chains in the subsystem have been traversed.

Recoup processing uses the FACSZ macro to simulate an environment so the records can be chased by the assigned processor on behalf of the owning environment.

When an item is recorded for every pool address in use in the system, the function of phase 1 recoup is complete.

Notes:

1. It is possible to have a record type in the system that can be referenced from several sources (for example, a passenger name record, or PNR). Under this condition, phase 1 chain chasing can spend a great amount of time reprocessing these chains.
2. In an MDBF environment, records will be chased starting with the first subsystem user (SSU) in a subsystem and ending with the last SSU for that

subsystem. Subsystem common records will be chased only once, and subsystem unique records will be chased by the owning SSU.

3. In a loosely coupled environment, all records of a given type will be chased from the processor assigned to that type in the IRSCT. Processor common records are chased only once. Processor unique records will be chased for each owning processor from a single assigned processor using the FACZ macro to access the file copy of the processor unique fixed file record. This approach is also used for records that are defined as processor or I-stream unique.
4. Because it is difficult in a loosely-coupled environment to guarantee that all processors in the complex will be up and running in the recoup process, it is mandatory that pool file address pointers maintained in global fields, global records, and core resident records be chased from the file copy of the records. The only exception to this restriction is when the records or fields have been defined to the TPF system as synchronized data (for example, all processors and I-streams have exactly the same view of the data).
5. Recoup phase 1 will also access all TPFCS data stores defined for the subsystem on which recoup is running and recoup the data store internal system collections. Recoup phase 1 will also recoup user-created collections by using the TPFCS recoup indexes to chain chase PIDs and extract embedded PIDs and file addresses. Recoup processing uses information in the base TPFCS data store (TPFDB), which resides on the basic subsystem (BSS). Only collections belonging to data stores on the subsystem where recoup is running will be recouped.

Post-Chain Chase Processing

After all record IDs have been chain chased, recoup does the following:

- Switches released address (FC33) records and the RTA tape to distinguish between pool addresses that were released before recoup chain chase processing starts and pool addresses that were released during chain chase processing. Any pool addresses that are released by applications during chain chase processing are recorded on new FC33 records and a new RTA tape.
- Captures the long-term file pool directory records onto the phase 3 captured SONRI directory (#SONCP). This capture is done to provide a comparison of long-term file pool addresses given out between phase 1 and phase 3 of recoup.

Phase 2 Functions

When chain chase processing is completed on all processors, the primary processor automatically starts recoup phase 2 processing, which does the following:

- Merges the ID counts records from all processors into a total ID counts for all processors.
- Merges all pseudo directories from all processors into one common pseudo directory.
- Checks the integrity of the VFA-resident pseudo directories to ensure they are accurate.
- Adds ID counts to a historical database where as many as 10 runs of ID counts are stored.

Recoup Phase 3 Functions

Recoup phase 3 can be started when phase 2 is completed and recoup pseudo directory (#SONRPE) records have been created. Phase 3 functions are started by entering commands. Recoup phase 3 processing consists of the following:

- Erroneously available processing

- Lost address processing
- Rebuild processing
- Rollin (pool directory update (PDU)) processing.

Erroneously Available Processing

The ZRECP RESUME command starts erroneously available processing, which does the following:

- Saves SONRI and keypoint 9 in the event that they are needed for a fallback.
- Applies the get file storage (GFS) activity that has taken place during chain chase processing to the recoup rollin directory (#SONROLL).
- Identifies pool addresses that are erroneously available and adds them to the erroneously available address table (#BREATB8) so they can be displayed.

Note: When erroneously available processing ends, you can enter the ZRECP PROTECT command immediately to protect pool addresses that were found to be erroneously available from being used by the TPF system. If no pool addresses were found to be erroneously available, enter the ZRECP IGNORE command instead.

Lost Address Processing

The ZRECP PROTECT or ZRECP IGNORE command starts lost address processing, which does the following:

- Saves the PDU pseudo directory (#SONUP) if recoup phase 3 needs to be restarted or run again.
- Applies pool addresses that have been released since the start of recoup to the pool rollin directories.
- Identifies pool addresses that are lost and adds them to the lost addresses table (#BRLOTB8) so they can be displayed.

Note: When lost address processing ends, you can enter the ZRECP ADD or ZRECP DEL command to create an exclusion or inclusion table. These tables are used by rebuild processing to adjust the lost addresses that are rolled in.

Rebuild Processing

The ZRECP REBUILD or ZRECP NOREBUILD command starts rebuild processing, which does the following:

- ZRECP NOREBUILD builds the recoup rollin directory (#SONROLL), releasing the lost pool addresses that were identified during lost address processing.
- ZRECP REBUILD builds the recoup rollin directory (#SONROLL), releasing the lost pool addresses (adjusted by the exclusion or inclusion table) that were identified during lost address processing.
- Displays recoup activity.

Rollin Processing

The ZRECP PROCEED command starts rollin processing, which adds pool files to the online pool directory (#SONRI) by using the recoup rollin directory (#SONROLL).

Recoup Phase 4 Functions

Integrated online pool maintenance and recoup support merged the recoup phase 3 and phase 4 functions into what is now called phase 3. Recoup phase 4 no longer exists.

Recoup Phase 5–Phase 7 Functions

Recoup phase 5 is run as an optional phase to help programmers determine the cause of lost and erroneously available file pool addresses. Phase 5 is not considered to be part of the normal recoup run. It is started when you enter the ZRECP DUMP command when phase 3 is completed. Phase 5 reads an input tape (ADR) and, using selective file dump and trace (SFDT), causes all lost and erroneously available file pool records to be written to the (RTL) tape.

Recoup phase 6 and phase 7 then use the information gathered by phase 5 to create various summary listings. Phase 6 and phase 7 are run offline under MVS, and the summaries that they create are used to help recoup determine the causes of lost and erroneously available addresses. See “Phase 6 JCL” and “Phase 7 JCL” for sample JCL to run recoup phase 6 and phase 7.

Phase 6 JCL

```
****Replace library parameters with installation's library
   *LINK.LIBRARY* = library containing PPCP offline segment
//POST  EXEC  PGM=PPCP,REGION=512K,PARM=('TV,Y,TR.')
//* TV => STV OR NO PTV      RS => RTL CREATED BY PTV
//STEPLIB  DD DISP=SHR,DSN=*LINK.LIBRARY*
//SYSOUT   DD  SYSOUT=A
//SYSUDUMP DD  SYSOUT=A
//SYSABEND DD  SYSOUT=A
//PRDD     DD  SYSOUT=A
//DDSCTH   DD  DSN=&&SCRTH,DISP=(NEW,PASS),
//             UNIT=(SYSDA,,DEFER),SPACE=(TRK,(50,200))
//SYS000   DD  DSN=RTA.TAPE,UNIT=TAPE,LABEL=(,SL),DISP=(OLD,KEEP),
//             VOL=SER=XXXXXX
//
```

Phase 7 JCL

```
****Replace library parameters with installation's library.
   *LINK.LIBRARY* = library containing BPR0xx offline segment
//STEP1   EXEC PGM=BPR0xx,REGION=400K,PARM='SSID'
//*  The PARM specifies the Subsystem data required from the RTL tape.
//*  It is the 4-character subsystem name. If omitted, all data
//*  pertinent to BPR0 is reported upon.
//STEPLIB DD DSN=*LINK.LIBRARY*,DISP=SHR
//PRINT   DD SYSOUT=A
//TAPEIN  DD DSN=RTA.TAPE,DISP=(OLD,KEEP),UNIT=(TAPE,,DEFER),
//    LABEL=(,SL),VOL=SER=XXXXXX
```

Recoup Procedures by Phase

The following lists the recoup procedures by phase.

Pre-Phase 1

Pre-phase 1 consists of steps to complete before you run recoup on the TPF system. This phase is not considered to be part of a normal recoup run.

Notes:

1. Cycle your TPF 4.1 system to CRAS state or above.
2. To run multiprocessor recoup in a loosely coupled environment, you must define recoup descriptor container records as core resident across all TPF images.
3. In a loosely coupled environment, you can run pre-phase 1 recoup from any active processor.

To run pre-phase 1 recoup, do the following:

- Before you run recoup for the first time, allocate #BKMST and #BKWRK as miscellaneous records with a record type of #IBMM4 and a record ID of BK (for each subsystem). (In MDBF, assign these records as subsystem common).
- Ensure that all recoup descriptor container records have been defined as core resident for each subsystem.
- If this is the first time you are running recoup, or if recoup descriptors have been changed, assemble the recoup descriptors on the offline MVS operating system.
- If recoup descriptors were created or changed, enter the ZOLDR LOAD command to load the assembled descriptors.
- If new recoup descriptors were loaded to a TPF image, enter the ZRBKD command, specifying the MOVE parameter to move the assembled descriptors to the BKD load control record (BK0LC).
- Enter **ZRBKD DISP ALL** to ensure that all the descriptors used for recoup are loaded in the BKD load control record (BK0LC) in the order in which you want them to run.
- If TPFCS was not initialized previously on your TPF system, enter **ZOODB INIT** to initialize TPFCS recoup before running recoup for the first time.
- Enter **ZRECP SETUP** once on each subsystem in which you run recoup.
- Create TPFCS recoup indexes for any user-created collections that contain embedded references and associate the recoup indexes with the appropriate collections.
- Enter the ZPROT DSP command to display the processor that owns the POOL utility. If the POOL utility is not assigned to the primary processor from which recoup will be started, enter the ZPROT command to assign ownership of the POOL utility to that processor.
- Enter the ZRECP PROFILE command to display and make any necessary changes to your recoup run-time options.
- If you have the TPFDF product installed and want to log errors online, enter **ZRECP ELOG ON**.

Phase 1 Procedures

Before running recoup phase 1, consider the following:

- In a loosely coupled environment, all processors can be in 1052 or NORM state:
  - If recoup is started in 1052 state, you can cycle up the TPF system later.

    Note: Recoup phase 1 timeout processing is not active while the TPF system is in 1052 state. In other words, entry control blocks (ECBs) that are in an endless pool chain loop cannot time out when the TPF system is in 1052 state. However, the timeout processing of TPFCS data stores is still active.
  - If recoup is started or running in NORM state, you cannot cycle down the TPF system until phase 1 is completed or a looping ECB error will occur.
- System performance can be affected by the number of ECBs being used by recoup phase 1. You can enter the ZRECP LEVEL command any time during recoup phase 1 to change the number of ECBs that recoup phase 1 can have active at any one time.

To run phase 1 recoup, do the following:

1. Ensure that all processors are in the correct system state.
2. Enter the ZRECP START command to start recoup.

3. When the RECP0020A – MOUNT RCP TAPE FOR OUTPUT message is displayed, mount the RCP general tape as an active blocked output tape. The RCP general tape must be mounted in the first subsystem user of the subsystem. The default is the BSS.
4. When the RECP004CA – ENSURE STANDBY RTA AND ALT TAPE MOUNTED FOR PHASE 1 message is displayed, mount a standby RTA and an ALT tape. (Recoup causes an RTA tape switch to occur.)
5. If you want to display or change your recoup run-time options, enter the ZRECP PROFILE command now (before you enter the ZRECP RECALL command).
6. Enter **ZRECP RECALL** after the RCP, standby RTA, and ALT tapes have been mounted. This command makes the control program file all records in virtual file access (VFA) if the TPF system is higher than 1052 state.

   Notes:

   a. You can enter the ZRECP STATUS command at any time during phase 1 to display summary status information about recoup phase 1 processing.
   b. If you have the TPFDF product installed and error logging is on, you can enter the ZRECP ONEL command at any time during phase 1 to display information about any errors that were found during phase 1.
7. If you are running recoup on more than one processor, enter (from the primary recoup processor) the ZRECP START command, specifying the PROC parameter for each processor that is going to run recoup.
8. If you receive the RECP0012A – FIXED ERROR – RESPOND and the RECP0014A – DEFERRING TIL SEL RECOUP COMPLETE OR CONTINUE RESPOND message, have your system programmer correct the problem and selectively add a record ID to a recoup run. See “Selectively Add a Record ID to a Recoup Run”.
9. You will receive the RECP0016I – PHASE I COMPLETED message when recoup phase 1 is completed.

Selectively Add a Record ID to a Recoup Run

Fixed records that are found in error during recoup chain chase processing are noted on a prime computer room agent set (CRAS) console during recoup phase 1 processing. Selectively adding a record ID to a recoup run allows you to correct and reprocess these records. If, after correcting the errors, you want to reprocess the records in error, enter the ZRECP SEL command for each record that you want to reprocess. This allows the corrected records to be included in the recoup run.

Note: TPFCS persistent identifiers (PIDs) can be reprocessed in a similar way.

Phase 1 Restart

To restart phase 1, do the following:

1. Ensure that the TPF system is in the state in which you originally started.
2. Remount an RCP tape as active output. (The RCP tape must be mounted in the first subsystem user of the subsystem.)
3. Enter **ZRECP RESTART** on the primary processor.

Phase 2 Procedures

Recoup phase 2 starts and runs without operator intervention. Recoup phase 2 writes error information to the RCP tape and, depending on your run-time options and online error log settings, online log files.

Note: If you do not have the TPFDF product installed, you cannot use the online error log. You will have to process the RCP tape offline using the BRFA program. See “Broken Chain Report (BRFA) Program”.

You can find additional information on commands in *TPF Operations*:

- See the ZRECP PROFILE command for more information about run-time options.
- See the ZRECP ELOG command for more information about turning on online error logging.
- See the ZRECP ONEL command for more information about displaying errors found during chain chasing.

Chain chasing multiple records that point to the same pool address, from different processors, causes ID counts for those pool addresses to be incorrect, because the pseudo directories are processor unique. These double-counted addresses are reported during recoup phase 2. Before you run recoup again, change the descriptor or DBDEF macro statement for the identified record to force cross-chained databases to be chain chased on the same processor.

Broken Chain Report (BRFA) Program

For each of the possible error conditions determined by phase 1 recoup, separate listings are produced in phase 2. These listings are as follows:

1. Broken chains due to invalid ID
2. Broken chains due to invalid RCC
3. Broken chains due to software errors
4. Broken chains due to hardware error
5. Broken chains due to invalid control information

Note: Pool addresses with error types 1, 2, and 3 are set to available in the pseudo directories recoup builds if no valid reference to those addresses are found. Pool addresses with hardware errors and control information errors are not returned to the pool, and will remain in IN-USE state.

The phase 2 listings show the following information:

Reference From	Contains the record ID, RCC, and file address of the record that sought the next address in the chain that it was chasing.
Seek Address	Contains the record ID, RCC, and the file address of the record that was sought by the *referenced from* record.
Err	For ID or RCC errors only; contains the record ID or RCC of the record that was found by the *referenced from* record.

These listings can be in more than one part if all the error items could not fit in main storage at one time. A double asterisk in the left-hand margin shows that the line contains the total number of errors for that part of the listing. A single asterisk shows that the line contains the total errors for the record type shown.

The following shows an example of the broken chain report:

```
                                 BROKEN CHAINS DUE TO INVALID ID                                       PART   1 PAGE   1  01131
 ********************************************************************************************************************************
 *  REFERENCE FROM LOCATION       SEEK ADDRESS INFO       ERROR /  REFERENCE FROM LOCATION       SEEK ADDRESS INFO       ERROR  *
 *                                                              /                                                               *
 *  ID  RC FILE ADDRESS     DISP  ID  RC FILE ADDRESS     ID/RC /  ID  RC FILE ADDRESS     DISP  ID  RC FILE ADDRESS     ID/RC  *
 ********************************************************************************************************************************
 *  JA  00 00000000FC3C0001 0040  JB  00 00000000001492F1 FC10  /  JA  00 00000000FC3C0001 0028  JE  00 00000000001492ED FC10   *
 *  JA  00 00000000FC3C0005 0060  JB  00 00000000001492FD FC10  /  JA  00 00000000FC3C0005 0040  JB  00 00000000001492F9 FC10   *
 *  JA  00 00000000FC3C0005 0028  JE  00 00000000001492F5 FC10  /  JA  00 00000000FC3C0009 0040  JB  00 0000000000149311 FC10   *
 *  JA  00 00000000FC3C0009 0028  JE  00 000000000014930D FC10  /  JA  00 00000000FC3C000D 0060  JB  00 000000000014931D FC10   *
 *  JA  00 00000000FC3C000D 0040  JB  00 0000000000149319 FC10  /  JA  00 00000000FC3C000D 0028  JE  00 0000000000149315 FC10   *
 *
**    10 = NUMBER OF BROKEN CHAINS FOR ID TYPE  JA                                                                              *
 *
 *  JG  00 00000000FC3E0001 0200  KB  00 000000000827BA09 FC10  /  JG  00 00000000FC3E0001 0100  KA  00 000000000827BA05 FC10   *
 *  JG  00 00000000FC3E0001 0060  K8  00 00000000001493B9 FC10  /  JG  00 00000000FC3E0001 0050  K8  00 00000000001493B5 FC10   *
 *  JG  00 00000000FC3E0001 0040  K8  00 00000000001493B1 FC15  /  JG  00 00000000FC3E0001 0030  K8  00 00000000001493AD FC15   *
 *                                                                                                                              *
**     6 = NUMBER OF BROKEN CHAINS FOR ID TYPE  JG                                                                              *
 ********************************************************************************************************************************
```

A summary listing is also printed that gives the type of errors, the total number of errors per error type, and the number of parts in the listing for each error type.

The following shows an example of the summary listing:

```
                        RECOUP FILE ANALYSIS
****************************************************************************
*           ERROR TYPE            * TOTAL ERRORS * NO. OF PARTS IN LISTING *
****************************************************************************
*                                 *              *                         *
* INVALID RECORD ID'S             *          5   *           1             *
*                                 *              *                         *
* INVALID RCC'S                   *          0   *           0             *
*                                 *              *                         *
* SOFTWARE ERRORS                 *          0   *           0             *
*                                 *              *                         *
* HARDWARE ERRORS                 *          0   *           0             *
*                                 *              *                         *
* INVALID CONTROL INFORMATION     *          0   *           0             *
****************************************************************************
```

Phase 3 Procedures

Phase 3 can be started and run in 1052 or NORM state. It should, however, remain in the state that it was started until it is completed.

To run phase 3 recoup, do the following:

1. Once recoup phase 2 has ended (online recoup displays a PHASE 2 COMPLETED message), enter **ZRECP RESUME**.
2. When you receive the RECP ENTER ZRECP PROTECT OR IGNORE TO CONTINUE message, enter **ZRECP PROTECT** or **ZRECP IGNORE**.

   Notes:

   a. The ZRECP PROTECT command immediately protects pool addresses that were found to be erroneously available from being used by the TPF system. If no pool addresses were found to be erroneously available, enter the ZRECP IGNORE command instead.
   b. You can enter ZRECP DISPLAY ERRON to display pool addresses that were found to be erroneously available.
3. When you receive the RECP BUILD LOST ADDR EXCLUSION TABLE NOW AND REBUILD DIRECTORIES OTHERWISE ENTER ZRECP NOREBUILD TO CONTINUE message, do one of the following:
   - Rebuild the directories as follows:

    a. If you want to add the record ID of a lost pool address to the exclusion or inclusion table, enter the ZRECP ADD command.

       Note: You can enter ZRECP DISPLAY LOST to display pool addresses that were found to be lost.
    b. Enter **ZRECP REBUILD** to rebuild the directories and continue recoup.
  - Enter **ZRECP NOREBUILD** to bypass rebuilding the directories and continue recoup.
4. When you receive messages indicating ACTIVITY DURING RECOUP, PSEUDO DIRECTORY COUNTS, and RECOUP ACTIVITY COUNTS, have a system programmer analyze the information to determine whether the directories created by recoup should be rolled in to the online system. Run recoup phase 5 for more detailed information. See "Phase 5 Procedures" for more information.
5. When you receive the DESTRUCTIVE SEGMENT RESPOND ENTER ZRECP PROCEED TO ROLLIN OR ZRECP SKIP/ZRECP ABORT TO BYPASS ROLLIN message, do one of the following:
   - Enter **ZRECP PROCEED** to roll in directories.

     Note: If you run recoup phase 3 in NORM state, ZRECP PROCEED does not roll in pool directories that are in the active or standby state, but writes these directories to released pool address (FC33) records and the RTA tape. These directories are processed during the next online pool update run. Therefore, the display of the pool counts is slightly lower than the display of the counts in the pseudo directories.
   - Enter **ZRECP SKIP** to skip the rollin.
   - Enter **ZRECP ABORT** to end recoup phase 3.

Phase 3 Restart

If an error occurs during recoup phase 3, you can fix the error and restart recoup phase 3 from the point of the error.

Note: Do not enter the ZRECP RESTART command to restart phase 3 if the TPF system IPLed after you entered the ZRECP PROCEED command because this can destroy the database. You must start recoup again from phase 1.

To restart phase 3 recoup, do the following:
1. Ensure the TPF system is in the same system state it was in when you entered the ZRECP RESUME command to start recoup phase 3.
2. Enter **ZRECP RESTART**.
3. Continue with the normal phase 3 procedures.

Phase 4 Procedures

Integrated online pool maintenance and recoup support merged the recoup phase 3 and phase 4 functions into what is now called phase 3. Recoup phase 4 no longer exists.

Phase 5 Procedures

Recoup phase 3 generated an online file or ADR tapes that contains the file addresses of lost and erroneously available file pool records. In a loosely coupled environment, you can run phase 5 from any of the active processors, but it must remain in the processor where it started and can only be accessed from that processor.

To obtain a selective dump of the file addresses of lost and erroneously available file pool records do the following:

- If lost and erroneously available file pool records were written to ADR tapes, do the following for each ADR tape:
  1. Mount the ADR tape as an active unblocked input tape on the online system.
  2. Enter **ZRECP DUMP** to write the first 32 bytes of each record whose file address is listed on the ADR tapes to RTL tapes.
- If lost and erroneously available file pool records were written to an online file, enter **ZRECP DUMP** to write the first 32 bytes of each record whose file address is listed on the online file to RTL tapes.

Notes:

1. When all the addresses are processed, status message RECP0000I is displayed.
2. If an unplanned shutdown occurs during ZRECP DUMP command processing, you ***must*** enter **ZRECP DUMP RESTART** after the TPF system IPL to continue the job. Restarting recoup phase 5 dump processing can cause recoup phase 7 output to show duplicated record IDs and incorrect counts.

The RTL tapes can subsequently be processed by the postprocessor (STPP) or summarized by record ID through the phase 3 output analyzer (phase 6 and phase 7).

The following headers and message formats accompany the record dumps on the RTL tape:

```
***START OF RECOUP SELD - LOST ADDRESSES FIRST***
***RECOUP SELD - START OF ERRONEOUSLY AVAIL ADRS***
```

Phase 6 and Phase 7 Procedures

Phase 6 consists of postprocessing the RTL tapes created by phase 5 under the postprocessor (STPP). Phase 7 consists of a BPR0 offline program that processes the same RTL tapes and creates listings for more information. This program summarizes the output of the phase 3 run. Lost addresses are summarized first and erroneously available addresses, if any, are summarized next. Use the sample JCL provided in “Phase 6 JCL” on page 109 and “Phase 7 JCL” on page 109 to run these phases.

Recoup Procedure for a Single Database

Sometimes you may not want to recoup your entire database. The following *selective database* recoup procedure allows you to process a single record ID, persistent identifier (PID), or data store. To run selective database recoup, enter the following:

1. **ZRECP START SEL**.
2. The ZRECP RECALL command, specifying the SEL parameter with a record ID, PID, or data store.
3. **ZRECP PROTECT SEL**.

Recoup Records and Structures

Certain other data is required during the various phases of recoup.

Recoup Keypoint Record (BK0RP)

Two copies of the recoup keypoint record are used by the recoup package. These are called the master and working copies. Both the master and the working copies reside in the fixed file area of the online system. The master copy is initialized by recoup phase 1 at the start of each recoup run and the working copy is initialized from the master copy. Both copies of the recoup keypoint record must reside in each subsystem.

Recoup Data Store

Before recoup processing can be run on a subsystem, a TPFCS data store must be created for that subsystem by entering the ZRECP SETUP command. The data stores are automatically assigned the name IRCP*ssss*, where *ssss* is the name of the subsystem.

IBM Recoup Scheduling Control Table (IRSCT)

The IRSCT contains information that determines in what order TPF records and TPFCS data stores will be processed during recoup phase 1. This table is represented as a TPFCS binary large object (BLOB). The IRSCT is rebuilt every time a ZRECP SETUP command is entered, or when ZRECP RECALL is entered for recoup phase 1. You can display the contents of the IRSCT by entering the ZBROW DISPLAY command with the ELEMENT parameter specified with a collection name of IRSCT.

IBM Recoup Active Root Table (IRART)

An IRART contains information about the TPF records or TPFCS collections that are in progress. This information is used if a recoup restart occurs. The IRARTs are represented as TPFCS BLOBs and are reinitialized every time a ZRECP SETUP command is entered or when the ZRECP RECALL command is entered for recoup phase 1. One IRART exists in the recoup data store for a given subsystem for each processor defined and has the name IRART_*cpuid*, where *cpuid* is the ID of the processor.

User Exits

Recoup user exits have been provided to allow concurrent use with unique user recoup packages. See *TPF System Installation Support Reference* for more information about user exits.

Real-Time Disk Formatter

Use the real-time disk formatter (FMTR) to prepare all direct access storage devices for system use. (See *TPF Migration Guide: Program Update Tapes* for a list of supported devices.) For all devices, this program formats any specified track up to the maximum cylinder and track values specified for the particular device type.

This program is run under MVS system control. It is assembled and run as a normal MVS job.

Method of Operation

The formatter examines an input card for errors and determines the desired size and number of records per track. It then converts the starting and ending disk addresses (CCCCCHH) to binary and builds the count fields for the track. The count fields are then written to the disk. All I/O to the disk is performed by EXCP and tested for errors. If this is not the last track specified in the ending address of the input card, the track number is incremented and the count fields are built and written again. When the ending address is reached, another card is read in and the previous procedure is repeated. When the END card is read, a listing that contains the location, size, and duplication factor of every track on the pack is sent to the printer. The listing also includes the address of the VTOC and the addresses of any tracks that were not formatted. If you want a status report without formatting the pack, use an END card with the device type specified so that you read, but do not write, to the disk. An end-of-job message is then sent.

If the same type of record will be defined in 2 or more noncontiguous areas on disk, you will need 2 or more cards to describe these areas.

Size and duplication indicators on the input card are used to set the last 3 bits of the disk record numbers to the correct value for small and large record types (2KB and 4KB records are compatible with MVS record numbers). (See the information about macro usage conventions in *TPF General Macros* for details about the meaning of these bits.)

Note: For 3390 storage devices in native mode and newer devices, the last 3 bits of the disk record numbers are not used as control bits. They are part of the physical record number. Therefore, the duplication indicator (N or D) is ignored by FMTR and can be omitted from the formatter control cards for 3390 devices.

A 3390 installed in 3380 emulation mode appears to the TPF system as a 3380 and must be coded as if it was a physical 3380. Therefore, the duplication factor is required for emulated 3380s.

Table 13 shows the number of records per track and the number of cylinders for TPF-supported DASD.

Table 13. Record Allocation on TPF-Supported DASD

Device	Large Rec/Trk	Small Rec/Trk	2KB Rec/Trk	4KB Rec/Trk	Number of Cylinders
3350	15	34	8	4	555
3375	25	46	14	8	959

Table 13. Record Allocation on TPF-Supported DASD (continued)

Device	Large Rec/Trk	Small Rec/Trk	2KB Rec/Trk	4KB Rec/Trk	Number of Cylinders
3380	30	53	18	10	2655
3390	33	55	—	12	3339
9345	27	47	—	10	2156

Note: The number of cylinders that are shown for 3380 and 3390 devices reflects the highest density devices currently supported. Your devices may have fewer cylinders than those listed.

2KB (IPC) records are not supported on 3390 and newer devices. The TPF system does not use 2KB records or support SIPC data sets on general files.

Input

The input for the real-time disk formatter consists of data control cards and formatter control cards.

This program will format 1 disk at a time, with the volume label of the pack to be formatted being specified by a standard DD card used in MVS JCL. The volume serial number on the pack can be any standard label that you want (6 alphanumeric digits), and the data set name can be any valid data set name allowed under MVS.

Although this program formats 1 disk pack at a time, many packs can be formatted at once by running different jobs, calling the formatter at the same time. The amount depends on the number of nonallocated disk modules and the amount of main storage that is allocated to the user in the particular MVS system being used.

All disk packs to be formatted must have first been initialized by MVS initialize disk program ICKDSF. This is required to properly label and format the first track and to place the volume table of contents on the pack.

Data Control Cards

You need 3 data control cards to run this program.

Data Device Name	Use
FMTDD1	Read input card (SYSIN)
FMTDD2	Printer status report (SYSOUT)
FMTDD3	Pack to be formatted

MVS Job Control Cards and Sequence of Deck (EXEC)

```
//FORMAT       JOB    (Accounting Information)
//STEP1        EXEC   PGM=FMTRXX (XX-LINKED VERSION)
//SYSPRINT     DD     SYSOUT=A
//FMTDD2       DD     SYSOUT=A
//FMTDD3       DD     DSNAME=PARSRCDS,UNIT=SYSDA,VOLUME=SER=(PARCRY),
                      DISP=(NEW,KEEP),SPACE=(TRK,7654)
//SYSUDUMP     DD     SYSOUT=A
//FMTSNP       DD     SYSOUT=A
```

```
//FMTDD1       DD     *

                      ------- Formatter Control Cards -------
/*
```

When you format a disk pack for the first time, the disposition of the data set is new (DISP=(NEW,KEEP)). When the packs are being reformatted, the disposition is old (DISP=(OLD,KEEP)). See *MVS/ESA JCL Reference* for additional information about coding MVS job control cards.

Formatter Control Cards

The following shows the format for the formatter control cards. The scale at the top of the figure indicates the column position for each parameter.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
 FMT     s     d cccchh   cccchh     dddd      comments
```

Where:

s is the record size, which can be one of the following:

- **S** Small size record (381 bytes)
- **L** Large size record (1055 bytes)
- **4** 4KB record (4096 bytes)
- **2** 2KB record (2048 bytes); only supported for SIPC data sets on general files; for migration purposes only.

d is the duplication factor, which can be one of the following:

- **D** Duplicate area
- **N** Nonduplicate area.

Note: The duplication indicator is ignored for 3390s in native mode and newer devices and, therefore, can be omitted for these devices. In addition, the duplication indicator is not applicable for 2KB or 4KB records.

ccccchh
is the first and last track to be formatted. For example, 0012208 to 0013814 specifies that cylinder 122, head 8 to cylinder 138, head 14 will be formatted.

dddd
is the device type to be formatted; for example, 3390. This is required only on the first card. See the *TPF Migration Guide: Program Update Tapes* for a list of the supported devices.

Note: 3390s installed in 3380 emulation mode must be coded as 3380s.

comments
Columns 50–80 can contain optional comments.

You can use as many control cards as necessary. Ranges of tracks can overlap; in this case the last card that affects a particular track determines the format. Control cards do not need to be in order by cylinder and head, but it will reduce the run time if they are. The FMT END card must be the last card in the control deck.

The following shows some sample input:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
 FMT     L     D 0000004   0001019     3390      LARGE DUP
 FMT     S     D 0001100   0018919               SMALL DUP
 FMT     2       0019000   0019819
 FMT     4       0019600   0019919
 FMT     END
```

Notes:

1. Cylinder 0, track 0 is reserved for the volume label and IPL records and cannot be formatted.
2. At least one other track must be set aside for the volume table of contents (VTOC); this track cannot be formatted either. If the VTOC track is specified in the formatter control card, a message is sent to the printer stating that the track is a VTOC track and cannot be formatted. The program then continues to format the next track specified in the formatter control card.
3. An abend will also occur if you specify a cylinder higher than the cylinder limit for the pack indicated on the control card.

Output

The output of the real-time disk formatter is a disk pack that is formatted for online use. In addition, a status report is provided for the disk pack that was formatted with information about the type of records on each track. The following is an example of a status report:

```
******  DISPLAY OF FORMATTED DISK   ******
                                       CYL-HEAD TO  CYL-HEAD
      LARGE NONDUPLICATE              000 001      009 007
      4K RECORD                        009 008      010 001
      LARGE NONDUPLICATE              010 002      044 028
      VTOC TRACK                       044 029      044 029
      NOT FORMATTED                    045 000      554 029
SMALL RECORD:   0381 BYTES      LARGE RECORD:   1055 BYTES      2K RECORD:
2048 BYTES      4K RECORD:      4096
END OF JOB
```

FACE Driver and Offline Interface (DFAD)

Use the FACE driver and offline interface (DFAD) to print all record-type ordinal-number combinations and their associated FACE generated file addresses. DFAD also provides an interface to the file address compute program (FACE) so that offline programs can obtain an address for a specific record type-ordinal number combination.

DFAD is run under MVS control. You can run this program using one of the following methods.

- Assemble and run the program as a normal MVS job. This is referred to as *file print* in the remainder of this chapter.
- Call the program from an assembly language program that is running under MVS control. This is referred to as the *offline interface* in the remainder of this chapter.

DFAD must be assembled against the STCEQ and the SYSEQC macros, which were created for the FACE table (FCTB) being processed.

Note: DFAD does not return information for FARF6 or unique record types.

File Print

The DFAD file print routine uses MVS job control language (JCL) statements as input and produces a listing that contains the record type, ordinal number, and associated FACE generated file address.

The following shows an example of the JCL statements needed to run the file print routine:

```
//DFAD    EXEC PGM=DFAD40,PARM='ALL,nn'
//JOBLIB  DD DSN=ACP.LINK.RLSE40.BSS,DISP=SHR
//SYSOUT  DD SYSOUT=A
//DFADPRT DD SYSOUT=A
```

Where:

- `nn` is a 2-character FCTB version.

  Note: If you do not specify the PARM parameter, the program prompts the operator with the following message to supply the required version number.

  ```
  ENTER FCTB TABLE VERSION, (FF TO END JOB)
  ```
- The JOBLIB statement defines where the FCTB is located.
- The DFADPRT statement defines where the output report will go.

DFAD runs a FACE-type call for each record type index in the range 0 through the highest record type defined. If the record type is valid to FACE, DFAD prints the address returned from FACE in the current dispense format for every valid ordinal number for that record type. The output listing contains the following information for valid record types:

- The 8-character record type from the corresponding entry in the SYSEQC macro
- The ordinal number in hexadecimal format
- The file address in hexadecimal format.

DFAD prints a single line message for invalid record types.

The following shows an example of the output listing.

```
          ORDNL  RETURNED           ORDNL  RETURNED           ORDNL
   ID     NUMBER ADDRESS     ID     NUMBER ADDRESS     ID     NUMBER

 *      NEXT RECORD TYPE ATTEMPTED:   ID=004(        )
 RECORD TYPE NOT ALLOCATED/NOT IN USE,ID=004(        )

 *      NEXT RECORD TYPE ATTEMPTED:   ID=005(#SSSRI  )
 #SSSRI   000000 02800006  #SSSRI   000001 0280000E  #SSSRI   000002
 #SSSRI   000004 02800026  #SSSRI   000005 0280002E  #SSSRI   000006
 #SSSRI   000008 02800046  #SSSRI   000009 0280004E  #SSSRI   00000A
 #SSSRI   00000C 02800066  #SSSRI   00000D 0280006E  #SSSRI   00000E
 #SSSRI   000010 02800086  #SSSRI   000011 0280008E  #SSSRI   000012
```

File Print Errors

If the version of the FCTB specified is not available, the job ends with system completion code of 806.

If you specify PARM data that is less than 6 characters or if ALL is not the first parameter, the following message displays:

```
INSUFFICIENT OR ERRONEOUS 'PARM' DATA
```

A return code of 16 is placed in register 15 and the job ends.

Offline Interface

An offline (MVS) program that requires a file address can link or call DFAD. To do this, code the following statement in the MVS program:

```
CALL DFAD, (recid,ord,ret,nn)
```

Where:

recid is a halfword record type equate value.

ord is a fullword ordinal number.

ret is the FACE program output area. The 4-byte FACE output will be located at `ret`+4. This address can be aligned to any boundary.

nn is the 2-character FCTB version in EBCDIC form.

In this mode of operation no external (for example, print or console) communication is used.

DFAD loads the FACE table (FCTB) and gets a single file address. If no errors occur, a zero return code is set in register 15, and DFAD returns to the calling program with the file address placed at the location specified by the user.

Offline Interface Errors

If the version of the FCTB specified is not available, the job ends with system completion code of 806.

A return code is placed in R15:

0	Normal return
1	Invalid or not-in-use record type
2	Ordinal number greater than maximum
3	Unsupported unique record type
4	FARF6 splits are not supported

8 Calling sequence error.

TPF Transaction Services

TPF transaction services includes support for a transaction manager, resource managers, log manager, and recovery log to ensure a consistent view of the database. A consistent view of the database ensures that either all or none of the file changes have been completed; there is no such thing as a partially updated database.

The *transaction manager* (TM) provides a set of application program interfaces (APIs) for the application to define both the scope of a transaction as well as actions to be taken for the transaction. The TM coordinates resource managers and determines which resources are written to the recovery log at commit time and which resources are recovered at restart time. The TM is aware of which resource managers are participating in TPF transaction services processing as well as what transactions are active. The TM also understands both nested and suspended commit scopes and uses a set of XA functions, defined by the X/Open specification, to interface with the resource managers.

The following *resource managers* (RMs) supplied by IBM work with the TM to identify and harden (write to the recovery log) resources used by the application in a commit scope:

- TPF DASD: handles updates to the TPF database.
- TPF MQSeries: handles updates to queues.
- Pool support: manages pool file addresses.

During normal processing, RMs do the following:

- Record a recoverable event when a native resource manager request is received
- Write all recoverable events to the recovery log when an application program enters a commit request (either with the `tx_commit` C function or the TXCMC assembler macro)
- Harden the recoverable events after they have completed writing to the recovery log.

You can write your own RM provided that it adheres to the architected TM and RM interface. Adhering to the interface means that the user-written RM can participate in all TPF transaction services actions controlled by the TM and all log and recovery actions controlled by the *log manager* (LM).

The log manager controls the recovery log and recovery actions.

The *recovery log* is written to DASD and holds the data necessary to recover resources following a system failure without compromising the integrity of the database. The recovery log is defined as vertically allocated, processor unique, fixed file records. This permits the log to be *striped* (written one track at a time, indexing across all of the modules for the device type) across the TPF database, which gives the TPF system the ability to handle an arbitrarily large data rate.

Commit Scope Processing

The following information describes the file-type requests and MQSeries-type requests used for commit scope processing.

File-Type Requests

At a high level, file-type requests that are entered in a commit scope are cached by the TPF system in virtual file access (VFA). This affects the amount of storage used for VFA. When you enter a find-type request for a record that is written by a commit scope (or a higher-level commit scope if the commit scope is nested), the record is read from a special VFA buffer, called a *commit scope buffer*, instead of from a normal VFA buffer or from DASD.

Note: Commit scope buffers do not age out of VFA.

The commit control records that are used by the transaction manager are maintained in system work blocks (SWBs); this will increase the number of SWBs that are required in the TPF system.

DASD hold processing is recorded in the commit control records for use when a commit transaction (`tx_commit` C function or TXCMC assembler macro) or rollback transaction (`tx_rollback` C function or TXRBC assembler macro) is entered. The record hold table (RHT) is used to maintain the following:

- A commit-level hold (when an entry control block (ECB) is not currently holding a lock and the unlock was done in an uncommitted commit scope).
- An ECB-level hold (when an ECB holds a lock, whether the ECB is holding the lock in a commit scope).

Specifying this as an external is intended to minimize updates to customer tools that recognize the record hold table.

MQSeries-Type Requests

At a high level, all processor unique queues are maintained in system heap and SWBs to obtain extremely high access rates to TPF queues. To assist with the recovery of persistent messages after an IPL, the contents of all queues are checkpointed on a time-initiated basis to fixed file records. All updates made by using the MQPUT, MQPUT1, or MQGET functions between checkpoints are written to the recovery log.

All processor shared queues are filed on TPF collection support (TPFCS).

Recovery Log Support

Recovery log support is general in nature and is designed to be extendable to support recovery of additional resources such as message and tape processing. User resource managers can be written to allow application-specific resources to use the recovery log to become part of the recoverable transaction. The TPF system uses the recovery log with TPF transaction services processing.

The recovery log permits the TPF system to recover, over a software or hardware IPL, to the point of failure without compromising the integrity of the database. Recovery is to a point at which all TPF transaction services processing is either redone or not redone. Partial update conditions will not occur.

The recovery log is defined as vertically allocated 4-KB fixed file records residing on a single logical device type. The log does not span multiple device types. Each 4-KB record is in the standard TPF format; that is, each record contains a standard 8-byte header. The data area of each record is relatively unstructured and contains information relating to transaction manager and resource manager write requests.

Writing to the Recovery Log

Writing to the recovery log, which consists of the writing of control information and data, is done through the use of the WLOGC macro. A user exit is activated when WLOGC processing has been completed to give users the opportunity to access the data being written; for example, to write the information to tape.

Writing associated commit scope log records is a multiple write event, as the following describes:

1. This process is started with a commit transaction. The transaction manager directs all of the associated resource managers to prepare the data to be committed.
2. Each resource manager then enters WLOGC macros to write its data to the recovery log.
3. Once all of the resource managers (RMs) have completed preparing the data, the transaction manager (TM) writes a record, using the WLOGC macro, to indicate that the `xa_prepare` function has been completed.
4. The TM then directs all of the RMs to harden their data (write the data to the recovery log.)
5. When hardening has been completed, each resource manager enters a WLOGC macro to indicate that the data has been hardened.

Reading from the Recovery Log

Only the log manager (LM) can read from the recovery log.

Restarting from the Recovery Log

During restart (after an IPL), the recovery log is used to recall information to bring the TPF system back to the state it was in before the IPL occurred. The recovery log is used for any DASD writes or MQSeries requests that may have been in transition or lost.

Loosely Coupled Considerations and Log Takeover

The recovery log is defined as processor unique, enabling each processor in the loosely coupled complex to maintain and recover its own log without regard to the other processors. The recovery log is a viable entity as long as its processor is up and running. Under certain conditions, it is necessary to recover the log to preserve a consistent commit database.

- One of these conditions occurs during the IPL of a processor because of an outage. In this case, the log is recovered during restart (after the IPL).
- A second condition occurs when a processor is excluded from the complex because of a hardware problem or as a nightly collapse procedure. Under the latter condition, it is expected that the processor will not be immediately IPLed. Therefore, to preserve a consistent commit database, the log must be run as part of the processor deactivation process.
  - For a condition where a processor is excluded from the complex by way of a normal cycle down and deactivation, the ZPSMS command with the PROC DEACT parameters specified permits active ECBs to drain from the system. Once all activity has been completed, the log is in a state where no transactions exist that would need to be redone; the log is essentially empty. Log recovery does not need to be activated. When you IPL a processor from the deactivate state, the recovery log is reinitialized and any previous log information is considered to be not valid.

  - For a condition where a processor has stopped because of a hardware error and is not capable of running its own log, it is necessary for another processor to control the recovery. This process is known as *log takeover*. Log takeover occurs during processing of the ZPSMS command with the PROC FORCE DEACT x parameters specified.
- A third condition occurs during a catastrophic outage that involves the entire complex, where the ensuing IPL is of the destructive type (that is, IPL BYPASS). For this condition, the first processor to IPL successfully is responsible for recovering its own log and the log for each of the other processors.

TPF Collection Support

TPFCS Database Layout

TPF collection support (TPFCS) contains all the components that you need to access collections. The TPFCS database consists of user-defined data stores. In these data stores are collections. Each collection consists of elements. Elements can contain character data, binary data, structures, or references to other collections or TPF files. The TPFCS database will handle collection sizes from 0 to 2 147 483 647 elements.

To relate the concept to familiar terms, you might say that an element could be a record, a file could be a collection, and a related set of files could be a data store. All of these files put together are the TPFCS database.

Figure 8 shows the general layout of a TPFCS database:

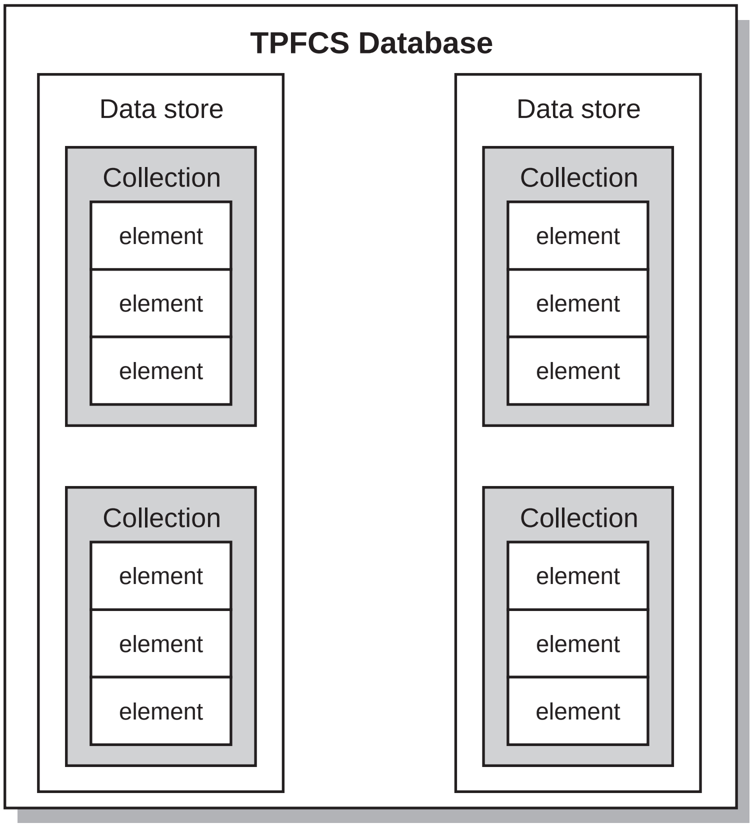


Figure 8. General Layout of a TPFCS Database

The following are some key points to remember about a TPFCS database:

- The basic subsystem (BSS) contains a directory for all data stores, including ones stored on other subsystems.
- The database contains user-defined data stores as well as multiple data stores.
- The database is anchored with two #IBMM4 records (one is a shadow, or copy, of the other).
- The database is initialized with the ZOODB INIT command.

Data Stores

A data store (DS) contains collections. A data store name is a maximum of 8 characters and can use printable special characters (for example: `!`, `&`, `%`, and `*`). Some data stores are created when TPFCS is initialized; others are user-defined and user-named by entering the ZOODB command (see “Maintaining TPFCS” on page 147).

The following are some important key points to remember about data stores:

- A TPFCS database can have multiple data stores. A data store provides the capability to divide a TPFCS database as shown in Figure 8 on page 129.
- Data store names must be unique across the entire TPFCS database, but the data store itself is subsystem unique and applications on other subsystems cannot access the data.
- Each defined data store contains collections. Any nontemporary collection with a persistent identifier (PID) can be stored in a data store.
- Some collections are automatically created and assigned a name when the data store is defined. These collections are used internally by the system or are available for use by the application:
  - DS_USER_DICT, an application dictionary. You can store whatever you want in this dictionary (for example, PIDs). See “Data Store Application Dictionary” on page 139 for more information.
  - DS_SYSTEM_DICT, a system dictionary. The system dictionaries are for use by TPFCS system code.
  - DS_BROWSE, a browser dictionary with a list of names associated with defined collections for use with the browser utility.
  - DS_INVENTORY, an inventory collection that contains the PIDs of all assigned collections.
  - DS_DELETED, a deleted PID collection that contains the PIDs of all collections scheduled to be deleted.
  - DS_RECOUP, a recoup repository containing the TPFCS recoup indexes that describe the layouts of collections for use by recoup.
  - DS_RESTARTLOG, a collection that contains information for TPFCS to use when an IPL occurs.
- Data stores can be created with different options, including whether or not an inventory collection should be automatically maintained (for tracking), and whether or not a collection delete request should be processed immediately or delayed 48 hours.

Collections

The TPFCS database provides basic support for the following types of collections (whether they are persistent or temporary):

- Arrays (single dimension)
- Bags
- BLOBs (binary large objects; also known as byte arrays)
- Key bags
- Key sets
- Key sorted bags
- Key sorted sets (also known as dictionaries)
- Keyed logs
- Log
- Sequence
- Sets

- Sorted bags
- Sorted sets.

TPFCS uses the collection manipulation routines to maintain its own collections, just as any application would maintain collections. By using this support, it is possible for TPFCS database users to access and retrieve information from the collection without having to retrieve the entire collection.

The following are some key points to remember about collections:

- All persistent collections are assigned a PID.
- A collection is created using a data definition (DD) defined for a particular data store (DS).
- Collections are created and deleted dynamically by using the `TO2_create...` and `TO2_deleteCollection` APIs, respectively.
- Long-term persistent collections reside on DASD in 4-KB long-term pool records.
- Short-term persistent collections reside on DASD in 4-KB short-term pool records.
- Temporary collections reside in the private heap storage of the entry control block (ECB) and overflow to short-term pool directories.
- The maximum collection size is as follows:
  - BLOBs: 2 GB (where 1 GB equals 1 073 741 824 bytes)
  - Others: 2-G elements (where 1 G equals 1 073 741 824 elements).

Elements

An *element* is a subunit of a collection. The following are some key points to remember about elements:

- Elements consist of any data type including binary strings and can hold references to other collections (PIDs) or TPF files.
- Elements with unformatted data can be stored in collections.
- Elements can be added, located, updated, or removed using TPFCS APIs (see *TPF Application Programming* for more information).

Collection Support

Figure 9 on page 132 shows an overview of the collection class hierarchy. The dashed boxes represent abstract classes and the solid boxes represent concrete classes (collection types) available with TPFCS. Solid lines between the classes represent logical inheritance of attributes and methods.

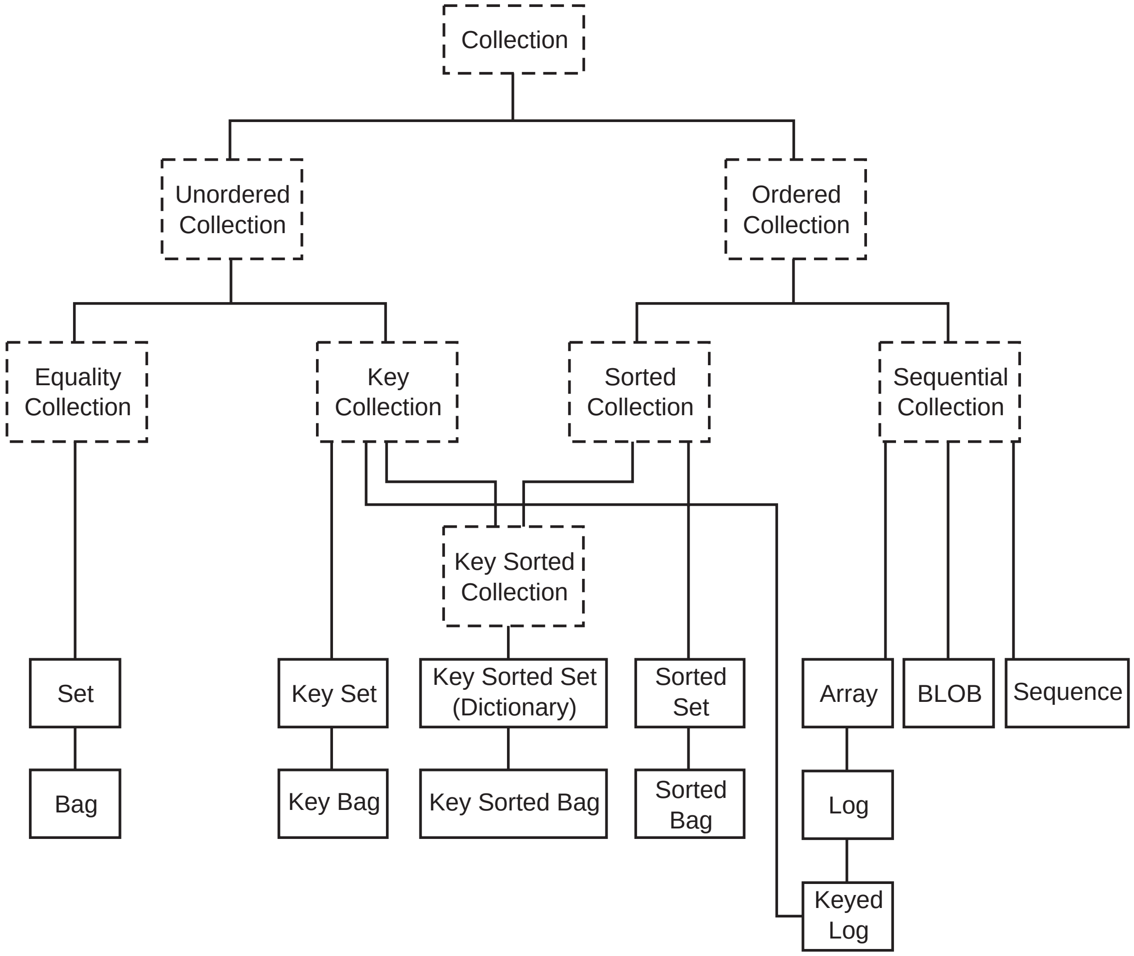


Figure 9. TPFCS Abstract Class Hierarchy

The following tables summarize collection support and can help you select the appropriate collection type for your application.

Table 14. Collection Support Summary - Part 1

Collection	Keyed	Duplicate Keys	Duplicate Values	Sort Fields	Element Equality
Array	No	N/A	Yes	No	No
Bag	No	N/A	Yes	No	Yes
BLOB	No	N/A	Yes	No	No
Key Bag	Yes	Yes	Yes	No	No
Key Set	Yes	No	Yes	No	No
Key Sorted Bag	Yes	Yes	Yes	No	No
Key Sorted Set	Yes	No	Yes	No	No
Keyed Log	Yes	No	Yes	No	No
Log	No	N/A	Yes	No	No
Sequence	No	N/A	Yes	No	No
Set	No	N/A	No	No	Yes
Sorted Bag	No	N/A	Yes	Yes	No

Table 14. Collection Support Summary - Part 1 (continued)

Collection	Keyed	Duplicate Keys	Duplicate Values	Sort Fields	Element Equality
Sorted Set	No	N/A	No[1]	Yes	No
Note: [1] The sort field must be unique, but the rest of the data value may be nonunique.					

Table 15. Collection Support Summary - Part 2

Collection	Ordered	Accessed	Added	Updated	Deleted
Array	By position (index)	By position (index)	At the end	By position (index)	N/A (zeros/elements)
Bag	Randomly	By value	Randomly	N/A	By value
BLOB	By position (index)	By position (index)	At the end	By position (index)	N/A (zeros/elements)
Key Bag	Randomly	By key	Randomly	N/A	By key
Key Set	Randomly	By key	Randomly	N/A	By key
Key Sorted Bag	By key	By key	By key	By key	By key
Key Sorted Set	By key	By key	By key	By key	By key
Keyed Log	By arrival	By key; by arrival position	At the end, with a wrap	N/A	N/A
Log	By arrival	By arrival	At the end, with a wrap	N/A	N/A
Sequence	By arrival; by position	By position (index)	At the end, by position	By position (index)	By position (index)
Set	Randomly	By value	Randomly	N/A	By value
Sorted Bag	By field	By field	By field	N/A*	N/A*
Sorted Set	By field	By field	By field	N/A*	N/A*
Note: * Elements in this collection can only be updated or deleted using cursors.					

Table 16. Collection Support Summary - Part 3

Collection	Maximum Key Length	Maximum Element Length[1]	Fixed Length Elements Required	Variable Length Elements Supported
Array	N/A	4000 bytes	Yes	No
Bag	N/A	248 bytes	No	Yes
BLOB	N/A	1 byte[3]	No	Yes
Key Bag	248 bytes	1000 bytes	No	Yes
Key Set	256 bytes	1000 bytes	No	Yes
Key Sorted Bag	248 bytes	1000 bytes	No	Yes
Key Sorted Set	256 bytes	1000 bytes	No	Yes
Keyed Log	256 bytes	4000 bytes	Yes	No
Log	N/A	4000 bytes	Yes	No
Sequence	N/A	4000 bytes	No	Yes
Set	N/A	256 bytes	No	Yes
Sorted Bag	248[2] bytes	1000 bytes	No	No
Sorted Set	256[2] bytes	1000 bytes	No	No

Table 16. Collection Support Summary - Part 3 (continued)

Collection	Maximum Key Length	Maximum Element Length[1]	Fixed Length Elements Required	Variable Length Elements Supported
Notes: [1] The element length does not include the length of a key, but it does include the length of a sort field. [2] The 248- and 256-byte lengths shown here are for sort fields, not keys. [3] A BLOB can also be considered to have a single element that is 2 GB.				

Collection Examples

This section lists TPFCS collections and provides an example of a potential application for each collection type. The examples can help you to understand the characteristics, behavior, and the overall capabilities of the collections.

Array: An *array* is an ordered collection of elements with no key. Element equality is not supported. The array elements are a fixed length and are accessed by relative position (index) in the array starting with index 1.

An example of using an array is a program for keeping track of seat assignments on an airplane. The program addresses array elements by using the seat number as an index to the corresponding entry in the array. When the seat is assigned, the name of the passenger is entered into the entry using the seat number as the index. You can determine the name of the passenger assigned to a specific seat by accessing the entry of the array indexed by the seat number. However, you cannot determine empty seats except by iterating through the array and testing for an empty entry.

Bag: A *bag* is an unordered collection of elements with no key. Nonunique elements having the same value are supported. This collection allows duplicate elements. The elements can have a maximum length of 248 bytes. A request to add an element that already exists will add the element again.

An example of using a bag is a program for tracking supplies in a medical supply room. Each time you spot a supply in the medical supply room, you enter the name of the supply into the collection. If you spot a supply twice during an observation period, the supply is added twice because a bag supports nonunique elements. You can locate the name of a supply that you have observed and you can determine the number of observations of that supply; however, you cannot sort the collection by supply (because a bag is an unordered collection). To sort the elements of a bag, use a sorted bag collection instead.

BLOB: A *BLOB*, sometimes referred to as a byte array, is a special array collection of elements with an element size of 1 byte. Element equality is not supported. The BLOB elements are addressed by relative byte address (RBA) starting with byte 1. With BLOBs, multiple operations can be performed on ranges of elements and data can be read in its entirety or part by part.

An example of using a BLOB is a program for keeping track of a patient’s X rays. The X ray must be available to be read or updated in its entirety or part by part. Each patient X ray is stored in the data store as a BLOB and can be directly addressed.

Key Bag: A *key bag* is an unordered collection of elements that have a key. Multiple copies of the same key are supported. The element value associated with the key is not relevant and the key is not part of the element data.

An example of using a key bag is a program that tracks assignments of dormitory suites. The element key is the number that is printed on the back of each key. Each element also has data members for the student name, student ID number, dormitory location, and so on. When you arrive at college, you are given one of the available keys, and your name, student ID number, and the dormitory location are entered into the collection. Because a given number on a key may appear on several keys, the program allows the same key number to be added to the collection many times. When you return a key because you are leaving the dormitory, the program finds each element whose key matches the serial number of your key and deletes one such element that has your name associated with it.

Key Set: A *key set* is an unordered collection of elements that have a unique key. Element equality (where all data members of both elements are equal) is not supported, and only elements with unique keys are supported. The key is not part of the element data. Requests to add an element whose key already exists result in an error return.

An example of using a key set is a program that allocates rooms to patrons checking into a hotel. The room number serves as the key of the element and the patron's name is a data member of the element. When you check in at the front desk, the clerk pulls a room key from the board and enters the number of the key and your name into the collection. When you return the key at checkout time, the record for that key is removed from the collection. You cannot add an element to the collection that is already present because there is only one key for each room. If you attempt to add an element that is already present, the function returns an error to indicate that the element was not added because of duplicate keys.

Key Sorted Bag: A *key sorted bag* is an ordered collection of elements that have a key. Elements are sorted according to the value of their key field, but the key is not part of the element data. Element equality is not supported while nonunique elements are. The maximum element size is 1000 bytes and the maximum sort field is 248 bytes.

An example of using a key sorted bag collection is a program that maintains a list of zip codes (for marketing purposes), sorted by the number of people in each zip code. The key is the zip code. You can add an element whose key is already in the collection (because two people can have the same zip code), and you can generate a list of people sorted by zip code; however, you cannot locate a person except by their key because a key sorted bag does not support element equality.

Key Sorted Set: A *key sorted set*, sometimes referred to as a *dictionary*, is an ordered collection of elements that have a key. Elements are sorted according to the value of their key field but, the key is not part of the element data. Element equality is not supported and only elements with unique keys are supported. Requests to add an element whose key already exists result in an error return.

An example of using a key sorted set is a program that keeps track of canceled credit card numbers and the individuals to whom they are issued. Each card number occurs only once and the collection is sorted by card number. When a merchant enters a customer's card number into a point-of-sale (POS) terminal, the collection is checked to see if that card number is listed in the collection of canceled cards. If the card number is found, the name of the individual is shown and the merchant is given directions for contacting the credit card company. If the card number is not found, the transaction can proceed because the card is considered to be valid. A list of canceled cards is printed out each month, sorted by card number, and distributed to all merchants who do not have an automatic POS terminal installed.

Keyed Log: The elements in a *keyed log* collection are ordered by arrival sequence; when the collection is full, the collection wraps and starts overlaying elements at the start of the collection. The keyed log consists of an element and key, where the key is a field contained in the element, and is not part of the element data. The keyed log assumes that the elements are a fixed length and are accessed by relative position (index) in the keyed log starting with index 1. The key can be used to access the collection to retrieve the element. There is no order to the keys of the collection.

Element fields other than the key field may be duplicated. If the key field is not unique, the add to the collection request will fail with an error return code. The elements can be a maximum length of 4000 bytes. The maximum key length is 256 bytes. For a keyed log collection, the first element is always the oldest entry still in the collection and the last element is the last to be added to the collection. Elements cannot be removed from a keyed log collection. The keyed log collection supports both access by position and by element key.

An example of using a keyed log collection is a program for tracking an account closing balance over the last 90 days. The keyed log collection would allow key access to specific dates over the 90 days being tracked. After 90 days, the collection is full and will begin overlaying transactions at the start of the collection.

Log: The elements in a *log* collection are ordered by arrival sequence; when the collection is full, the collection wraps and starts overlaying elements at the start of the collection. For a log collection, the first element is always the oldest entry still in the collection and the last element is the last to be added to the collection. The log assumes that the elements are a fixed length and are accessed by relative position (index) in the log starting with index 1. Elements cannot be removed from a log collection. The elements can have a maximum length of 4000 bytes.

An example of using a log collection is a program for tracking an account closing balance over the last 90 days. After 90 days, the collection is full and will begin overlaying transactions at the start of the collection.

Sequence: A *sequence* is a collection of elements with no keys. The elements are ordered by the arrival of requests. Element equality is not supported, while nonunique elements are supported. The type and value of the elements have no effect on the behavior of the sequence collection. Elements can be added and deleted from any position in the collection, can be retrieved or replaced. A sequence collection does not support element equality or a key.

An example of a sequence collection is a program that maintains a list of the words in a paragraph. The order of the words is obviously important; you can add or remove words at a given position, but you cannot search for individual words except by iterating through the collection and comparing each word to the word you are searching for. You can add a word that is already present in the collection because a given word may be used more than once in a paragraph.

Set: A *set* is an unordered collection of elements with no key. Element equality is supported and the values of the elements are relevant. Only unique elements are supported. A request to add an element that already exists is ignored.

An example of a set is a program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise and selects 10 items at random whose price is below a threshold level. Each item is then added to the set. The set does not allow an item to be added if it is already present in the collection, ensuring that a customer does not get two samples of a single product. The set is not sorted and elements of the set cannot be located by key.

Sorted Bag: A *sorted bag* collection is an ordered collection of elements with no key. Element equality is not supported, while nonunique elements are supported. Elements are sorted according to the value of their sort field, which is part of the element data. Nonunique elements are supported and a request to add an element that already exists will add the element again. The maximum element size is 1000 bytes and the maximum sort field is 248 bytes.

An example of using a sorted bag collection is a program for entering observations on the types of stones found in a riverbed. Each time you find a stone on the riverbed, you enter the mineral type of the stone into the collection. You can enter the same mineral type for several stones because a sorted bag collection supports nonunique elements. You can search for stones of a particular mineral type and you can determine the number of observations of stones of that type. You can also display the contents of the collection, sorted by mineral type, if you want a complete list of observations made to date.

Sorted Set: A *sorted set* collection is an ordered collection of elements with no key. Elements are sorted according to the value of their sort field, which is part of the element data. Element equality is not supported, while unique elements are supported; and a request to add an element that already exists will result in an error. The maximum element size is 1000 bytes and the maximum sort field is 256 bytes.

An example of using a sorted set collection is a program that tests numbers to see if they are prime. Two complementary sorted sets are used: one for prime numbers and one for nonprime numbers. When you enter a number, the program first looks in the set of nonprime numbers. If the value is found there, the number is nonprime. If the value is not found there, the program looks in the set of prime numbers. If the value is found there, the number is prime. Otherwise, the program determines whether the number is prime or nonprime and places it in the appropriate sorted set. The program can also display a list of prime or nonprime numbers beginning at the first prime or nonprime number following a given value because the numbers in a sorted set are sorted from smallest to largest.

Data Definitions

A data definition (DD) is an integral part of a collection definition. It provides characteristics for collections in a data store.

The following are some key points to remember about data definitions:

- Data definitions are data store unique. The same data definition name can be used with multiple data stores.
- Data stores may have multiple data definitions.
- Data definitions are used to assign the following record identifiers (IDs) to collections:
  - Data record ID
  - Key (index) record ID
  - Directory record ID.

Note: These record IDs and their attributes ***must*** be defined in the record ID attribute table (RIAT). Record IDs are used for services such as identifying virtual file access (VFA) candidates.

- Data definitions are used to specify shadowing, which indicates whether a second copy of each collection exists (see “Shadowing” on page 144).
- Data definitions are used to assign recoup indexes to collections. For more information about TPFCS recoup, see “TPFCS Recoup” on page 149 and *TPF Operations*.
- Data definitions can be defined, changed, displayed, or deleted by using the ZOODB command (see *TPF Operations* for more information).
- Data definitions are specified when the collection is created.
- Each collection has only one data definition.
- If a data definition is changed, only collections created after that will be affected. The definition for a particular collection is fixed once the collection is created.

Property Service

The TPFCS database supports a property service for persistent collections that are created. *Properties* are essentially typed named values that you can dynamically associate with an already existing persistent collection. Once these properties are defined, they can be named, their values can be set and obtained, their access modes can be set, and they can be deleted. These operations can be performed by using either the ZBROW PROPERTY command or the TPFCS property APIs.

The typed named values are:

Typed Name Value	Description
TO2_PROPERTY_CHAR	1-byte value.
TO2_PROPERTY_LONG	4-byte value.
TO2_PROPERTY_DOUBLE	8-byte value.
TO2_PROPERTY_STRING	C character string (maximum 256 bytes).
TO2_PROPERTY_STRUCT	Structure (maximum 1000 bytes).

The property service defines four mutually exclusive property modes:

NORMAL	The property can be read, changed, or deleted (there are no restrictions).
NOCHANGE	The property can only be read and deleted. It cannot be changed.
NODELETE	The property can be read or changed but it cannot be deleted. A NODELETE property mode is deleted only when the target collection is deleted.
READONLY	The property can only be read. It cannot be changed or deleted. A READONLY property mode is deleted only when the target collection is deleted.

Collection Lifetimes

TPFCS provides the following collection lifetimes:

Persistent long-term
Exists beyond the life of the creating ECB, resides on DASD in 4-KB long-term pools, and can survive a re-IPL. TPFCS reuses its own long-term pool

addresses so applications do not flush through pool records too quickly, causing the TPF system to run out of long-term pool records.

TPFCS reuses long-term pool records by maintaining a table for each subsystem. Each table is large enough to hold 100 file addresses. When TPFCS decides to return a long-term pool record, rather than release it to the system, it stores the file address of the record in the appropriate table if the following conditions are met:

- Recoup is not active
- There is no open commit scope for the ECB under which TPFCS is running
- The table is not already full.

If any of these conditions are not met or if the record is a short-term pool, the record is released to the system and its address is not stored in the table. When TPFCS needs to obtain a new long-term pool record, before allocating one using the GETFC macro, it looks in the appropriate table for a pool record it can reuse and then removes that entry from the table. Additionally, when TPFCS recoup starts, it releases all file addresses from each subsystem table and removes them.

Persistent short-term

Exists beyond the life of the creating ECB, resides on DASD in 4-KB short-term pools, and can survive a re-IPL. Persistent short-term collections are distinguished from persistent long-term collections by assigned record IDs and their RIAT attributes. Because the collection resides in short-term pools, it will be deleted when the short-term pools are recycled.

Temporary

Resides in the private heap area of the ECB and overflows to short-term pools. It is deleted when the ECB exits because the private heap area of the ECB is reclaimed by the system when the ECB exits. In order for the resources to be reclaimed correctly by the system, the collection should be explicitly deleted by the application before the ECB exits. This allows any overflow to short-term pools to be released and returned to the short-term pool directories for possible reuse.

The lifetime for a collection is established when the collection is created and cannot be subsequently changed. A collection lifetime is specified by selecting the appropriate collection creation API and data definition.

Deleting Collections

When you define a data store using the ZOODB DEFINE command, you can specify the disposition of deleted collections; that is, whether the collection deletion will be delayed 48 hours or whether the collection will be deleted immediately. If the deletion is delayed, collections can be reclaimed until the actual deletion takes place using the ZBROW COLLECTION command or the `TO2_reclaimPID` function. See “Maintaining TPFCS” on page 147 and the *TPF C/C++ Language Support User’s Guide* for more information.

Entering the ZBROW COLLECTION command with the EMPTY parameter specified on a DS_DELETED collection forces delayed deletions to occur for the particular data store. See *TPF Operations* for more information about the ZBROW COLLECTION command.

Data Store Application Dictionary

Each data store contains a dictionary automatically created by TPFCS that is available for use by the application. This dictionary is known as the data store application dictionary and is assigned the name DS_USER_DICT. This dictionary is accessed by establishing an environment for the target data store and using the `TO2_...DSdict...` type functions. The dictionary uses EBCDIC keys of 64 bytes with data elements of 1000 bytes.

One suggested use for the application dictionary is to place the PIDs of data store anchor collections in the dictionary and to assign a symbolic name to each one as the key.

Note: When PIDs are stored in collection elements, a recoup index must be established and associated with that collection.

The application dictionary of TPFDB, the base TPFCS data store, can be accessed with a special set of functions (`TO2_...TPF...`). This allows all applications access to a common dictionary without needing to establish an environment for a particular data store. A possible use for this dictionary is to associate applications with data stores.

Note: The data store system dictionaries, accessed with the `TO2_...DSsystem...` and `TO2_...TPFsystem...` type functions, are used by the TPFCS system and are not intended to be used by applications.

Cursors

A *cursor* is a nonpersistent internal structure associated with a collection that is used to reference an element in the collection. Cursors are used for the following:

- To iterate through collections

  The cursor provides methods that allow an application program to move through a collection one element at a time without needing keys or indexes. See “Using Iterative Operations over Collections” on page 141 for more information.
- To prevent concurrent updating of a collection

  To prevent concurrent updating of a collection while you are accessing elements in the collection, you can establish a locking cursor. When a locking cursor is created for a particular collection, a lock is placed on the entire collection so that only the ECB that created the cursor can update the elements in the collection. See “Concurrency Controls” on page 142 for more information.
- To access collection elements in different predetermined orders

  For certain types of collections, cursors allow you to either access a particular element using a field other than the primary key or sort field, or to iterate through a collection in an order other than that determined by the primary key or sort field. These tasks involve the use of alternate key paths, which can only be used by cursors. See “Using Key Paths” on page 141 for more information.
- To improve processing efficiency

  It is more efficient to access multiple elements in a collection by using a cursor. Most TPFCS functions are *atomic*; that is, the element is read into storage, managed, and removed from storage for each function call. This type of processing requires much overhead. With cursors, once a collection is read into storage, it remains there for the life of the cursor. The cursor is temporary and must be deleted by the application program or when the ECB exits.

Cursors can be used with any collection. If you use the cursor APIs to add or remove elements, cursor positioning remains valid unless an error occurs. If you use the collection (non-cursor) APIs on collections that have a cursor associated with them to add or remove elements, cursor positioning might not be valid. For more information about cursors, see *TPF Application Programming*.

Using Iterative Operations over Collections

Iterating over all or some elements of a collection is a common operation. TPFCS gives you two methods of iteration:
- Using cursors
- Using the `TO2_allElementsDo` function together with your own function.

Some ordered collections, such as an array or sequence collection, are easy to traverse because elements can be referenced by a numerical position. However, other ordered collections, such as key sorted sets and sorted bags, cannot be easily traversed without advance knowledge of the key and sort values. Unordered collections, such as bag and key set, also cannot be easily traversed. Using one of the two TPFCS methods of iteration allows all elements to be visited exactly once. For unordered collections, there is no defined order in which the elements are visited, but each element is still visited exactly once. If you add or remove elements from a collection while you are iterating over a collection (except by using the cursor doing the iteration), all elements may not be visited once. See *TPF Application Programming* for information about adding and removing elements.

Using Key Paths

TPFCS supports primary and alternate key paths for the following persistent keyed and sorted collections:
- Key bags
- Key sets
- Key sorted bags
- Key sorted sets
- Sorted bags
- Sorted sets.

Note: Sorted bag and sorted set collections do not have primary keys; they only have sort fields.

A key path establishes the order in which elements will be sequentially accessed when iterating through a collection using a cursor. There are two types of key paths: primary and alternate. When a collection is first created, the primary key path of the collection is used by default for searching and accessing data. After the key path has been defined, TPFCS will automatically update it whenever the collection is updated. Therefore, collections with alternate key paths will take longer to update because they have to maintain the internal structures of every key path.

You can use the `TO2_setKeyPath` function to override the default setting or any previous `TO2_setKeyPath` function calls to specify an alternate key path. A maximum of 16 alternate key paths (in addition to the primary) can be defined for each collection. When the `TO2_setKeyPath` function call is issued, the position of the cursor must be reestablished by using one of the positioning functions, such as `TO2_first`.

See *TPF Application Programming* for information about adding, using, and removing alternate key paths or the *TPF C/C++ Language Support User’s Guide* for information about the `TO2_first` or `TO2_setKeyPath` function.

Database Integrity

This section provides information about services that maintain TPFCS database integrity.

Database Access

There are no restrictions on which users can access the TPFCS database. Database access is denied only if the data store you requested does not exist. An application program informs the TPFCS database that you want to access the database. The TPFCS database assigns a token and saves any information required by the TPFCS database to communicate with you. Once you are known to the TPFCS database, you have access to all of the collections in the database. The TPFCS database has no service that restricts your access to only a subset of collections. If you are allowed to access the database, the TPFCS database then allows access to all the collections in the database.

Concurrency Controls

TPFCS provides three levels of concurrency control:

- None (using nonlocking cursors)
- Optimistic (using update sequence counters)
- Pessimistic (using locking cursors)

None (Nonlocking Cursor)

The first type of concurrency uses a nonlocking cursor to read elements in a collection without creating any type of interlock on the target collection. The same ECB or other ECBs can still issue requests to update the collection while the cursor is associated with the collection. Because the information can change underneath the cursor, the cursor may not iterate over all elements, it may iterate over some elements more than once, or it may no longer be valid. For this reason, a nonlocking cursor is considered a *dirty-read* cursor (see “Dirty-Reader Protection” on page 143). A subsequent request to lock the collection by the same ECB or another ECB will be successful and an exclusive lock will be placed on the collection by the new cursor. However, the nonlocking cursor will still be able to read elements in the collection. Nonlocking cursors are created by using the `TO2_createCursor` API. For more information, see *TPF Application Programming*.

Optimistic Concurrency (Update Sequence Counter)

To provide additional database integrity for those collections that allow element updating, an update sequence counter is stored in each element in a given collection and optimistic concurrency is enforced on the element during update processing. Optimistic concurrency allows you to read a collection and update it without exclusive access to the collection. When an element is to be updated, it must first be read from the collection with the returned sequence counter saved. The functions used to update elements in a collection require that the application provide the expected value for this counter. TPFCS increments the counter whenever a collection element is updated. If the collection is updated by some other user, your current update request will fail because the value passed by the application as input to the function does not match the update sequence counter embedded in the element. The collection must be retrieved again for a successful update to take place.

Pessimistic (Locking Cursor)

Pessimistic concurrency uses a locking cursor to create an exclusive lock on the collection. Write operations will only be allowed for the ECB creating the locking cursor. Any attempt by the same ECB to create another concurrent locking cursor on the collection will be rejected, and a request by another ECB to create a locking cursor will be forced to wait until the first ECB either deletes its locking cursor or exits. Locking cursors are created by using the `TO2_createReadWriteCursor` API. For more information, see *TPF Application Programming*.

Comparing Cursor Types

The following summarizes TPFCS operations involving the two different types of cursors:

- When a nonlocking cursor is used on a collection, the same ECB can:
  - Perform noncursor reads and writes
  - Perform cursor reads
  - Create other nonlocking cursors
  - Create a locking cursor.

A different ECB can also do all of the above.

When a locking cursor is used on a collection, the same ECB can:

- Perform noncursor reads and writes
- Perform cursor reads and writes
- Create other nonlocking cursors.

The same ECB cannot create another locking cursor. A different ECB can do all of the above and it can also create another locking cursor, although the request will be deferred until the original locking cursor is deleted.

Dirty-Reader Protection

TPFCS provides dirty-reader protection by using the FILNC macro to file records in a sequence to make sure that an updated or new record is on the DASD surface before filing the record that points to the updated record. This is done to ensure that another user who is attempting to read the collection using a nonlocking cursor will be able to follow a whole chain. If the updates were filed in the wrong order, it might be possible for the reader to follow a chain with holes in it or the chain could point to records that were not even part of the collection; for example:

> Record A contains a pointer to record B and now TPFCS has to insert record C between record A and B. If TPFCS filed record A with the new pointer to record C before filing record C, it would then be possible for a reader to read record A and then attempt to read record C before it had been filed. To prevent this, TPFCS dirty-reader protection will make sure that record C is filed first by using the FILNC macro, and then record A will be filed.

For more information about the FILNC macro, see *TPF General Macros*.

TPF Transaction Services

TPFCS uses TPF transaction services to create a commit scope on behalf of the caller for all update requests with the exception of cursor update requests. The application is responsible for commit scopes when using cursors. TPF transaction services will then either commit or roll back the change depending on the success of the request.

Note: To determine which TPFCS functions use commit and rollback protocols, see the *TPF C/C++ Language Support User's Guide*.

The TPFCS database will also use commit and rollback protocols for maintaining its own structures in a consistent state. Therefore, an API that does not use commit and rollback on behalf of the caller may still activate the system commit and rollback protocols on behalf of the TPFCS database. Prime examples of this are most cursor functions that cause a collection to be changed.

TPF Transaction Services with Cursors

When using cursors, you should create a commit scope before creating a locking or nonlocking cursor on a collection. When using cursors, the TPFCS database maintains the state of the collection cached in storage. This can cause problems because there is no communication between the TPFCS database and the transaction processor of the system. Therefore, if you issue a commit or rollback request, the collection may not be committed or rolled back. You must delete the cursor before the commit or rollback request is issued. Otherwise, there is a possibility that some updates will not be committed or rolled back.

TPF Transaction Services with Dirty-Reader Protection

Using dirty-reader protection in a TPF commit scope can cause problems because of the way TPF transaction services supports FILNC macro requests in a loosely coupled system. TPF transaction services maintains a unique copy of every record filed using the FILNC macro. This is done so that TPF transaction services can honor the FILNC interface and make sure that the record has been filed before any following records that might have a pointer to the record. Therefore, every time a record is filed using the FILNC macro, TPF transaction services will add a copy of the record to the commit scope. For example, if the record is filed 500 times using the FILNC macro in the commit scope, there will be 500 copies of the record in the commit scope. This can cause TPF transaction services to exhaust its commit buffers. For more information about dirty-reader functions and programming considerations, see the *TPF C/C++ Language Support User's Guide*.

Shadowing

TPFCS provides an option on the `TO2_create` functions that allows you to specify that the collection is to be *shadowed*. When TPFCS shadows a collection, two copies of the collection are maintained: the prime and the shadow. TPFCS does this by allocating two DASD records for every record assigned to the collection. TPFCS will read one of the two records and will file all updates to both records. If the initial read fails, TPFCS automatically attempts to read the copy. When using normal TPF duplicate records for the collection, shadowing means that TPFCS is actually maintaining four copies of the data.

The shadow support can be set for a collection in one of two ways: The `TO2_create` type call that creates the collection can have the shadow option specified, or the specified data definition (DD) can have shadowing set on in it. If either the create call or the DD specifies shadowing, TPFCS activates shadowing for the created collection. Once a shadowing characteristic is created for a collection, it cannot be changed.

Note: Shadowing is only supported for nontemporary persistent collections. The shadow option will result in an error if specified for a temporary collection.

To specify shadowing for a collection, enter the ZOODB DEFINE or ZOODB CHANGE command with the DD and SHADOW parameters specified. For more information about the ZOODB DEFINE or ZOODB CHANGE command, see *TPF Operations*.

Validation

TPFCS validation involves using the ZBROW command to perform a check of a subset of collection structures to ensure that they are built correctly. Errors are detected, isolated, and reported to the calling program, such as the ZBROW utility. A validation report will be returned to the user. This report will contain information about the types of errors found and the file addresses involved. The report will be sorted by error identifier. Each type of error will have a unique error identifier that indicates the severity of the error (generally, the lower the error identifier, the higher the severity). It is up to you to take additional actions to diagnose and repair the problems.

Reconstruction

Reconstruction involves correcting a subset of the fields in a control record of the collection and the chains that it anchors. Such reconstruction would be necessary if data becomes inaccessible as a result of data or control record corruption, logic errors, I/O errors, or a user error. The ZBROW RECONSTRUCT command is available to reconstruct these internal data and control structures from the beginning. It is up to you to make additional repairs by using the ZAFIL command.

Database Archives

This section provides information about the TPFCS database archive services.

External Device Support

External device and archiving support provide interfaces that will allow you to read and write data from external devices such as tape, general data sets, communication devices, and any other devices supported by the TPF 4.1 system. The initial support will be for tape only. As part of the external device support for tape, functions have been added that will interface with the tape robotics to allow access to tapes without operator intervention. Through the support for tape and tape robotics, external device support will be used to provide an archiving function that will be used by TPFCS to archive collections. Through the use of tape robotics and archiving, TPFCS will be able to move collections in and out of the TPF 4.1 system as required.

Archiving Support

For archiving, the application calls the `TPFxd_archiveStart` function. The `TPFxd_archiveStart` function will return a token that the application passes to TPFCS as a parameter on the `T02_capture` or `T02_restore` function. Once the request has completed successfully, the application can continue with another `T02_capture` or `T02_restore` function request until all required collections have either been archived or restored from the archive devices. The application will then issue a `TPFxd_archiveEnd` function request. Therefore, the application program determines if the TPFCS request is for archiving or to just restore another external device.

Capture and Restore Support

TPFCS provides functions to write and retrieve collections to external storage. By using these TPFCS functions and the archiving support of the external device support, collections will be able to be moved in and out of archived storage.

The `T02_capture` and `T02_restore` functions are used to write and retrieve collections from the external device using the external device support functions.

When an application decides to capture or restore a collection, the application must choose which external device will be used for the function. The application starts a capture or restore by issuing a `TPFxd_externalStart` function call with the required values. This function call will return a token that the application passes to TPFCS as a parameter on the `T02_capture` or `T02_restore` function. Once the request has been completed, the application issues a `TPFxd_externalEnd` function call to inform the external device support that it is done.

When TPFCS captures a collection to external storage, it will precede the collection with a capture header object and it appends a capture trailer object to the captured collection. Therefore, the captured object will start with a header object and end with a trailer object. Between the two objects, there will be the actual collection being captured. The captured object will be written as a set of control information and 4-KB data records. For each set of data records, there will be a data trailer object to indicate the end of one set of data records from another set of data records. Figure 10 on page 147 shows an example of data on the external device.

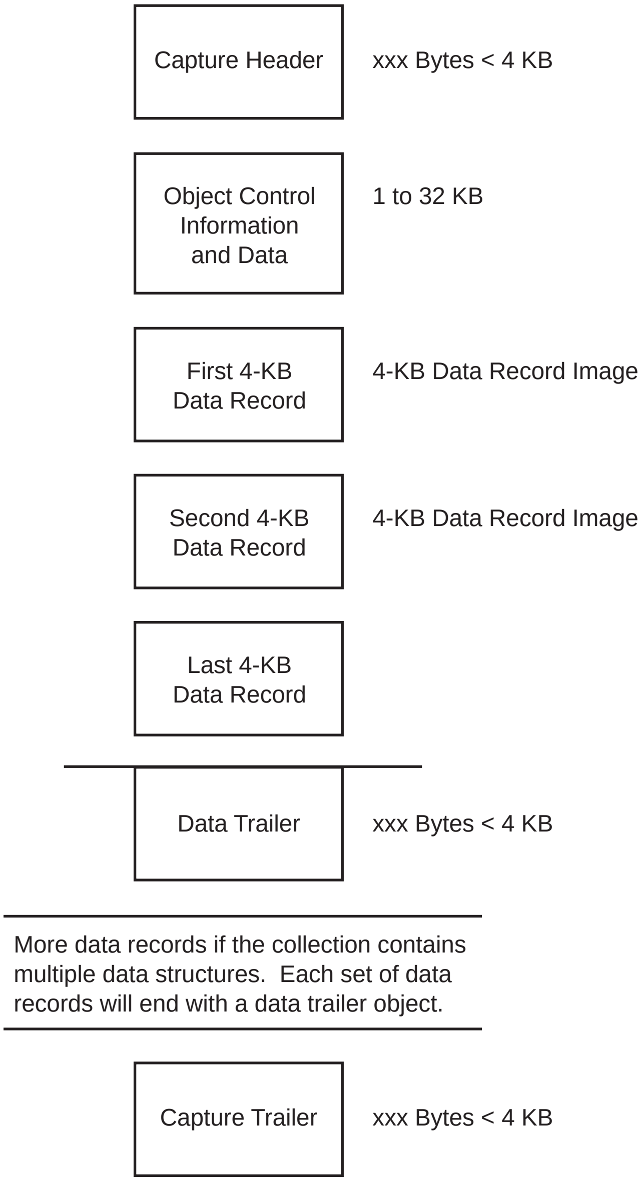


Figure 10. Data on an External Device

If the collection is small enough to fit in the object control information area, the capture trailer object will follow the object control information.

Maintaining TPFCS

This section provides information about the commands used to initialize and maintain the TPFCS database.

Note: Most of the functionality achieved by using the commands can also be implemented by writing applications that use the TPFCS APIs.

ZOODB Commands

The ZOODB commands are used to initialize the TPFCS database, define and display a data store or data definition, and change or delete a data definition. The following describes the ZOODB commands:

Function	Description
ZOODB CHANGE	Changes a data definition or data store.
ZOODB DEFINE	Defines a data definition or data store.
ZOODB DELETE	Deletes a data definition or data store.
ZOODB DISPLAY	Displays a data definition or data store.
ZOODB INIT	Initializes support.
ZOODB MIGRATE	Migrates a data store.
ZOODB RECREATE	Re-creates a data store.
ZOODB SET	Sets on or sets off the method trace table or set dump creation on a `TO2_getErrorText` function call.

Note: For more information about the ZOODB commands, see *TPF Operations*.

Initializing TPFCS

The ZOODB INIT command is a one-time-only message that is used to initialize TPFCS. If you attempt to initialize TPFCS again, an error message will be displayed.

After the ZOODB INIT command has been completed successfully, enter the ZOODB DEFINE command with the DS parameter specified to create user-defined data stores.

ZBROW Commands

TPFCS provides browsing support that allows classes, methods, and collections to be located, interrogated, validated, displayed, and dumped. This support is provided by using the ZBROW commands and TPFCS function calls.

The following describes the ZBROW commands:

Function	Description
ZBROW ALTER	Changes the contents or access mode of a specified collection.
ZBROW CLASS	Displays class name information.
ZBROW COLLECTION	Performs maintenance on a collection.
ZBROW DISPLAY	Displays information about a collection or its contents.
ZBROW KEYPATH	Adds, displays, or removes a key path.
ZBROW NAME	Alters or displays collection name information.
ZBROW PATH	Displays path information for a collection structure.
ZBROW PROPERTY	Alters or displays a property.
ZBROW QUALIFY	Qualifies ZBROW command requests for a data store.
ZBROW RECOUP	Manages recoup indexes.

See *TPF Operations* for more information about the ZBROW commands. See the *TPF C/C++ Language Support User’s Guide* for more information about the TPFCS C function calls.

TPFCS Recoup

Records that are used by TPFCS data stores are recouped as an extension of phase 1 of TPF recoup.

Because TPFCS has no knowledge of the contents of the data stored in a collection, this knowledge must be provided to TPFCS through another means so that collections that contain references to standard TPF files or to other collections can be recouped correctly. TPFCS can obtain information about the contents of the data by requiring the applications to describe the layout of data in a collection. Applications do this by identifying the displacements of any embedded 4- or 8-byte file addresses or PIDs.

General Approach

During phase 1 of TPF recoup, TPFCS recoup will recoup its own internal collections. However, any collections created by user applications will not be recouped unless they are explicitly made known to TPFCS by establishing one or more recoup indexes. A *recoup index* describes the location of PIDs and 4- or 8-byte file addresses embedded in all collections associated with that recoup index. An anchor collection, usually the data store application dictionary (named DS_USER_DICT) of the data store, refers to user-created collections, which refer to other collections, and so on. Every user-created collection must be referred to by an anchor collection or it will not be recouped. A recoup index must be created and associated with the anchor collection and any other collections that contain embedded references.

Recoup Indexes

Recoup indexes can be managed by an application or by using the ZBROW RECOUP command.

A separate index should be created for each of the different collection formats in your TPFCS database. A recoup index is unique to the data store. If necessary, a recoup index can be deleted.

After a recoup index is created, you must add one or more entries to it. (Previously added entries can be deleted from a recoup index if they are no longer needed.) Recoup index entries identify the location of file addresses or PIDs in the collection data. 4- or 8-byte file addresses embedded in TPFCS collections must be part of a traditionally chained TPF structure and must have a corresponding TPF recoup descriptor. File addresses of TPFDF records cannot be embedded in TPFCS collections.

There are two basic types of recoup indexes. The first type, known as an *homogeneous index*, is used when each of the elements in the collections associated with this index has the same format and can be recouped the same way. A homogeneous recoup index can be associated with any type of collection with the exception of BLOBs. The second type, known as a *heterogeneous index*, is used when the elements in the collections associated with this index cannot be recouped the same way because they have different formats. When adding entries to a heterogeneous index, you must specify which elements have embedded 4- or 8-byte file address or PID information (as well as the displacements in those elements where the file address or PID information is stored). The following restrictions exist:

- Heterogeneous recoup indexes can be associated with all collection types ***except*** for BLOBs.

- Heterogeneous indexes associated with keyed collections must access elements by key, and heterogeneous indexes associated with non-keyed collections must access elements by index.
- Partial keys cannot be used by heterogeneous recoup indexes to access collection elements.

Because BLOBs have a single element, special processing is required and the recoup index type indicator is ignored. Therefore, the recoup index associated with a BLOB is not considered to be either homogeneous or heterogeneous.

After the recoup index has been created, it can be associated with one or more collections if they have the same format for embedded 4- or 8-byte file addresses or PIDs. However, a single collection can only be associated with one recoup index at a time. You can also associate a recoup index with a collection when the collection is created by using an option list. You can also remove an association whenever you have determined that you no longer want TPFCS recoup to use a given recoup index to process the collection.

TPFCS will use the TPFCS recoup indexes to chain chase any embedded 4- or 8-byte file addresses or PIDs while processing a ZRECP RECALL command. You can also recoup individual collections by identifying a specific data store (DS). In test mode, use the ZRECP RECALL command with the SEL and DS parameters specified to recoup a specific collection. In production mode, use the ZRECP SELECT command with the PID parameter specified to recoup a specific collection when the collection has errors. To display the status of TPFCS recoup, use the ZRECP STATUS command. TPFCS recoup status messages are displayed approximately every minute, when recoup processing for a data store ends or when the ZRECP STATUS command is entered. See *TPF Operations* for more information about the ZRECP commands.

The following table lists the operations you can perform on a recoup index and identifies the API or parameter of the ZBROW RECOUP command you can use for each operation.

Table 17. Managing TPFCS Recoup Indexes

Operation	**API**	**ZBROW RECOUP Command Parameter**
Create a recoup index	`T02_createRecoupIndex`	DEFINE
Add an entry to a recoup index	`T02_addRecoupIndexEntry`	ADD
Associate a collection with a recoup index	`T02_associateRecoupIndexWithPID`	LINK
Display recoup indexes	None	DISPLAY
Delete a recoup index to collection association	`T02_removeRecoupIndexFromPID`	UNLINK
Delete an entry from a recoup index	`T02_deleteRecoupIndexEntry`	REMOVE
Delete a recoup index	`T02_deleteRecoupIndex`	DELETE

See the *TPF C/C++ Language Support User’s Guide* for more information about TPFCS APIs and *TPF Operations* for more information about the ZBROW RECOUP command.

Embedded 4-Byte File Address Information

If 4-byte file addresses are embedded in a collection, they must be stored as a 16-byte entry of type TO2_RECOUP_FA, which has the following format:

unsigned char format
Indicates the type of file address that is contained.

TO2_RECOUP_FRMT_FARF
Indicates that *fileAddr* is a 4-byte file address.

unsigned char reserved1
Reserved for future IBM use; must be set to zero.

unsigned char flag
Indicates whether this reference contains overflow records or other referenced records.

TO2_RECOUP_DSCR_NO
Indicates that this reference does not contain any overflow records or other referenced records.

TO2_RECOUP_DSCR_YES
Indicates that this reference contains overflow records or other referenced records. Create a unique group or index record set by entering the GROUP macro with the USE parameter specified with a value of TPFCS, and the IND parameter specified with a value of C and loaded as a recoup descriptor. See *TPF System Macros* for more information about the GROUP macro.

unsigned char recID[2]
Hexadecimal record ID.

unsigned char rcc
Hexadecimal record code check (RCC), or set to zero.

unsigned char ctl
Must be set to 0.

TO2_FARF_FA *fileAddr*
File address.

unsigned long reserved
Reserved for future IBM use; must be set to zero.

Embedded 8-Byte File Address Information

If 8-byte file addresses are embedded in a collection, they must be stored as a 16-byte entry of type TO2_RECOUP_XFA, which has the following format:

unsigned char format
Indicates the type of file address that is contained.

TO2_RECOUP_FRMT_FARF6
Indicates that *fileAddr* is an 8-byte file address.

unsigned char reserved1
Reserved for future IBM use; must be set to zero.

unsigned char flag
Indicates whether this reference contains overflow records or other referenced records.

TO2_RECOUP_DSCR_NO
Indicates that this reference does not contain any overflow records or other referenced records.

TO2_RECOUP_DSCR_YES
Indicates that this reference contains overflow records or other referenced records. Create a unique group or index record set by entering the GROUP macro with the USE parameter specified with a value of TPFCS, and the IND parameter specified with a value of C and loaded as a recoup descriptor. See *TPF System Macros* for more information about the GROUP macro.

unsigned char recID[2]
Hexadecimal record ID.

unsigned char rcc
Hexadecimal record code check (RCC), or set to zero.

unsigned char ctl
Must be set to zero.

TO2_FARF6_FA *fileAddr*
File address.

Embedded Persistent Identifier (PID) Information

If PIDs are embedded within a collection, they must be stored in the following format:

TO2_PID
The PID of the referenced collection.

Note: Recoup will ignore an embedded PID that is all zeros.

Sample TPFCS Recoup Applications

The following examples are intended as suggested uses for TPFCS recoup applications.

The following example creates recoup indexes:

```
/*********************************************************************/
/* Sample application to create recoup indexes                       */
/*********************************************************************/

#include <tpfapi.h>        /* Needed for TPF_regs structure          */
#include <c$to2.h>         /* Needed for TO2 API functions           */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main( void )
{
  TO2_ENV_PTR        env_ptr = NULL;     /* pointer to environment   */
  TO2_USER_TOKEN     userToken = 0;      /* user token               */
  char               applid[] = "RECOUP_INDEX_CREATOR            ";
  char               dsname[] = "TESTX.DS";

  TO2_PID            dsDictPID;          /* DS dictionary PID        */
  TO2_PID_PTR        dsDictPtr = &dsDictPID; /* ptr to DS dict PID   */
  char               IndexName[8];       /* Recoup index name        */
  TO2_RECOUP_TYPE    rc_type;            /* recoup index type        */
  TO2_RECOUP_CONTROL rc_ctrl;            /* recoup index control     */
  TO2_RECOUP_ENTRY_TYPE    rce_type;     /* PID or FA indicator      */
  TO2_RECOUP_ENTRY_ACCESS  rce_access;   /* index or key indicator   */
  char                     rce_token[8]; /* recoup index entry token */
  long                     rce_displ;    /* displ. to embedded ref.  */
  long                     rce_valLen;   /* length of collect. index */
  long                     rce_value;    /* collection index         */

/*********************************************************************/
/* Create TO2_environment                                            */
/*********************************************************************/
  if (TO2_createEnv(&env_ptr,&userToken,applid,dsname) == TO2_ERROR)
  {
     printf("ERROR: createEnv failed ! \n");
     exit(1);
  }

/*********************************************************************/
/* Set up the recoup index for the application dictionary collection.*/
/* If this index already existed, delete it first and recreate it.   */
/* Each entry in the dictionary collection will have a PID embedded  */
/* at a displacement of 0, so the collection is homogeneous.         */
/*********************************************************************/
  if (TO2_getDSdictPID(dsDictPtr, env_ptr) == TO2_ERROR)
  {
     printf("ERROR: getDSdictPID failed\n");
     TO2_deleteEnv(env_ptr);
     exit(1);
  }

  memcpy(IndexName, "APPLDICT", 8);
  TO2_deleteRecoupIndex(env_ptr, IndexName);

  rc_type = TO2_RECOUP_HOMOGENEOUS;
  rc_ctrl = TO2_RECOUP_CONTROL_NONE;
  if (TO2_createRecoupIndex(env_ptr, IndexName, rc_type, rc_ctrl,
       NULL, NULL, "Used with data store appl dictionary") == TO2_ERROR)
  {
     printf("ERROR: createRecoupIndex %s failed\n", IndexName);
     TO2_deleteEnv(env_ptr);
     exit(1);
  }

  memcpy(rce_token, "ENTRY001", 8);
  rce_type = TO2_RECOUP_ENTRY_PID;
  rce_displ = 0;
  rce_access = TO2_RECOUP_ACCESS_NOTUSED;
  if (TO2_addRecoupIndexEntry(env_ptr, IndexName, &rce_token,
                              rce_type, &rce_displ,
                              rce_access, NULL, NULL) == TO2_ERROR)
  {
     printf("ERROR: addRecoupIndexEntry for DS failed !\n");
  }

  if (TO2_associateRecoupIndexWithPID(dsDictPtr, env_ptr, IndexName)
                                                       == TO2_ERROR)
  {
     printf("ERROR: associateRecoupIndexWithPID for DS failed !\n");
  }

/*********************************************************************/
/* Create a recoup index for a heterogeneous collection.             */
/* Every collection that is associated with this recoup index must   */
/* have the following format:                                        */
/*   The second element has two embedded references; a file address */
/*       at displacement 16, and a file address at displacement 40.  */
/*   The third element has an embedded file address at displacement  */
/*       14.                                                         */
/* (Note: Minimal error checking is shown.)                          */
/*********************************************************************/
```

```
  memcpy(IndexName,"HETERO01",8);
  rc_type = TO2_RECOUP_HETEROGENEOUS;
  rc_ctrl = TO2_RECOUP_CONTROL_NONE;
  if (TO2_createRecoupIndex(env_ptr, IndexName, rc_type, rc_ctrl,
                            NULL, NULL, NULL) == TO2_ERROR)
  {
     if (TO2_getErrorCode(env_ptr) == TO2_ERROR_INDEX_EXISTS)
        printf("Index %8s already exists\n", IndexName);
     else
     {
        printf("ERROR: createRecoupIndex %s failed\n", IndexName);
        TO2_deleteEnv(env_ptr);
        exit(1);
     }
  }

  memcpy(rce_token, "ENTRY01A", 8);
  rce_type = TO2_RECOUP_ENTRY_FA;
  rce_displ = 16;
  rce_access = TO2_RECOUP_ACCESS_INDEX;
  rce_valLen = 4;
  rce_value = 2;
  TO2_addRecoupIndexEntry(env_ptr, IndexName, &rce_token,
                          rce_type, &rce_displ, rce_access,
                          &rce_valLen, &rce_value);

  memcpy(rce_token, "ENTRY01B", 8);
  rce_displ = 40;
  TO2_addRecoupIndexEntry(env_ptr, IndexName, &rce_token,
                          rce_type, &rce_displ, rce_access,
                          &rce_valLen, &rce_value);

  memcpy(rce_token, "ENTRY02A", 8);
  rce_displ = 14;
  rce_value = 3;
  TO2_addRecoupIndexEntry(env_ptr, IndexName, &rce_token,
                          rce_type, &rce_displ, rce_access,
                          &rce_valLen, &rce_value);

/*******************************************************************/
/* Delete TO2_environment                                          */
/*******************************************************************/
  TO2_deleteEnv(env_ptr);
  exit(0);
}
```

The following example associates recoup indexes with collections:

```
/*******************************************************************/
/* Sample application to associate recoup indexes with collections   */
/*******************************************************************/

#include <tpfapi.h>        /* Needed for TPF_regs structure          */
#include <c$to2.h>         /* Needed for TO2 API functions           */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tpfio.h>         /* Needed for getfc()                     */

void main( void )
{
  TO2_ENV_PTR        env_ptr = NULL;       /* pointer to environment */
  TO2_USER_TOKEN     userToken = 0;        /* user token             */
  char               applid[] = "APPLICATION_1                   ";
  char               dsname[] = "TESTX.DS"; /* data store name       */
  char               IndexName[8];         /* Recoup index name      */
  TO2_PID            pid;                  /* area for array PID     */
  TO2_PID_PTR        arpid_ptr=&pid;       /* pointer to array PID   */
  char               element[50];          /* array element          */
  long               element_length;       /* length of array element*/
  TO2_RECOUP_FA      fa_entry;             /* fileAddr entry in elem */
  char               recid[2];             /* fileAddr record ID     */
  unsigned int       file_addr;            /* TPF record file address*/
  struct stdhdr     *file2rec_ptr;         /* pointer to TPF record  */
  long               size;                 /* size of PID            */
  TO2_OPTION_PTR     optionListPtr = NULL; /* pointer to option list */
  char               colName[64];          /* collection name        */

/*********************************************************************/
/* Create TO2_environment                                            */
/*********************************************************************/
  if (TO2_createEnv(&env_ptr,&userToken,applid,dsname) == TO2_ERROR)
  {
     printf("ERROR: createEnv failed ! \n");
     exit(1);
  }

/*********************************************************************/
/* Create an array that is associated with the heterogeneous recoup  */
/* index created in the previous sample application.  The array will */
/* be an anchor collection in the data store, so add it to the data  */
/* store application dictionary.  Also, assign a browse name to the  */
/* array that is the same as the array's key in the application dict.*/
/*********************************************************************/
  memcpy(IndexName,"HETERO01",8);
  element_length = sizeof(element);
  optionListPtr = TO2_createOptionList(env_ptr,
                                       TO2_OPTION_LIST_CREATE,
                                       TO2_CREATE_RECOUP,
                                       TO2_OPTION_LIST_END,
                                       IndexName);
  if (TO2_createArrayWithOptions(arpid_ptr, env_ptr, optionListPtr,
                                 &element_length) == TO2_ERROR)
  {
     printf("ERROR: createArray failed !\n");
  }
  free(optionListPtr);

  memset(colName, 0x00, sizeof(colName));
  memcpy(colName, "ARRAY1", 6);
  size = sizeof(TO2_PID);
  TO2_atDSdictNewKeyPut(env_ptr, colName, arpid_ptr, &size);

  TO2_defineBrowseNameForPID(arpid_ptr, env_ptr, colName);

/*********************************************************************/
/* Add the 3rd element to the array.  This element has a TPF file    */
/* address embedded at a displacement of 14 bytes.  The TPF file has */
/* a standard TPF header and will contain forward references.        */
/*********************************************************************/
  recid[0] = 0xAB;
  recid[1] = 0x12;
  file_addr = getfc(D2, GETFC_TYPE0, recid, GETFC_BLOCK|GETFC_FILL,
                    GETFC_NOSERRC, 0x00);
  memset(&fa_entry, 0x00, sizeof(fa_entry));
  fa_entry.flag = TO2_RECOUP_DSCR_YES;
  memcpy(&fa_entry.recID, recid, 2);
  memcpy(&fa_entry.fileAddr, &file_addr, 4);

  file2rec_ptr = ecbptr()->ce1cr2;
  memcpy(file2rec_ptr,&array_entry.recID,4);
  filec(D2);

  memset(&element, 0x00, sizeof(element));
  memcpy(element,"DATA BEFORE FA",14);
  memcpy(&element[14], &fa_entry, sizeof(fa_entry));
  memcpy(element,"DATA AFTER FA",13);
  TO2_add(arpid_ptr, env_ptr, element, &element_length);

/*********************************************************************/
/* Delete TO2_environment                                            */
/*********************************************************************/
  TO2_deleteEnv(env_ptr);
  exit(0);
}
```

TPF Collection Support Database from a TPF System Perspective

The applications that use TPF collection support (TPFCS) can be written with minimal knowledge of the TPF database. However, system database administrators who maintain the TPF database need to understand, at some level, how TPFCS stores collections in the TPF database. Such knowledge is helpful in detecting database corruption and in taking appropriate action.

System programmers also will benefit from a high-level understanding of how TPFCS represents and stores collections in the TPF database. TPFCS uses pool records to represent and manage collections. These pool records often contain data that represents displacement pointers that are used when a portion of an associated collection is read into memory. If a defective application corrupts pool records that TPFCS uses, it is possible for the TPFCS program to either detect a severe error or, depending on the nature of how the data is corrupted, the TPFCS code itself could experience a CTL-3 or CTL-4 system error. Such exceptions would be caused by defective applications that have corrupted the TPFCS displacement pointers for a collection, and would not necessarily be caused by defects in product code.

The way that TPFCS stores collections physically in the TPF database differs vastly from how those collections appear to an application programmer who writes programs that create and use those collections. For example, even if a key sorted set (dictionary) collection is small enough for TPFCS to file all of its data elements and their keys in a single 4-K record, those data elements and their keys might very well be written in that record in random physical order. TPFCS would keep track of the conceptual order as seen by application programs that access that key sorted set (dictionary) by maintaining pointer fields (locators) that it also stores with the data and their keys. The existence of these locators is totally hidden from application programs that access the key sorted set (dictionary).

There is no set way in which TPFCS chooses to store a given collection in the TPF database. Even two collections of the same type and the same number of elements can be stored by TPFCS in different ways in the same TPF database.

The information presented in this chapter provides information about how TPFCS represents collections so that problems such as data corruption can be detected and corrected. While you might have to call your IBM representative if the tools such as TPFCS capture and restore of collections are not enough to correct data corruption, it is imperative that you read this section to be able to work with your IBM representative to reach your goals.

Object-Oriented Concepts

TPFCS has been implemented using an object-oriented design. While application programmers need no knowledge of object-oriented concepts to use the services TPFCS provides, a knowledge of general object-oriented concepts described in this document is required to understand how collections are stored in a TPF database.

An *object* is an entity consisting of data and functions that are available to manage the data. The terms *attributes* and *methods* are often used to refer to the data fields and functions, respectively. Objects are classified into different categories. Each category of objects is known as an *object class*, or simply as a *class*. An object class is merely a description that defines the exact format of the attributes (data) as well as the exact methods (functions) to be applied to the data. The attributes and methods together comprise each object of that class. Each specific, existing example of an object is said to be an *instance* of the object class. In discussing object-oriented design, the terms *object* and *instance* are often used interchangeably. Both refer to an actual example of a generic type that is described by a class. Each object or instance contains all the attributes and methods defined for its class. We commonly say that each instance ***contains*** those attributes and methods from its class.

Some object classes can be categorized and divided into more specific classes, just as a generic category can be divided into more specific types. The more generic class is commonly known as the *superclass* of the more specific classes. Likewise, each of the more specific classes is commonly known as a *subclass* of the more generic class. A given subclass, in turn, can be divided into additional, more specific subclasses. Each subclass definition is built on the definition of its superclasses, and can also specify a set of additional attributes and methods unique to the subclass. An object, which is an instance of a subclass, not only contains or inherits the attributes and methods defined for the subclass, but also inherits the attributes and methods defined for each of the more generic classes (superclasses) as well.

You can best understand how different objects inherit attributes from different classes by drawing an *inheritance tree*, which is a diagram that shows how different classes and their objects are related. Consider the following example of an inheritance tree:

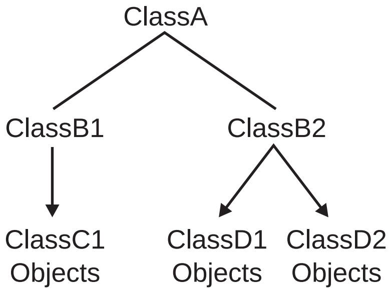


Figure 11. Class Inheritance Tree Example

In our example, ClassA is divided into two subclasses, ClassB1 and ClassB2. ClassB1 contains only one subclass (ClassC1), whereas ClassB2 is divided into two additional subclasses called ClassD1 and ClassD2 respectively. In our example, only the leaves of an inheritance tree represent classes for which specific objects or instances can exist. ClassA, ClassB1, and ClassB2 can be referred to as *abstract classes* because there are no objects belonging to those classes that do not also belong to an additional subclass.

All objects of ClassC1 not only inherit all of the attributes and methods defined for ClassC1, but also inherit the attributes and methods defined for ClassB1 and ClassA. Similarly, all ClassD1 objects inherit the attributes and methods defined for ClassD1, ClassB2, and ClassA; and all ClassD2 objects inherit the attributes and methods defined for ClassD2, ClassB2, and ClassA. Notice that, while all of the objects inherit the attributes and methods defined for ClassA, ***none*** of the objects inherit from ***all*** of the classes. For example, a ClassD1 object does ***not*** inherit attributes and methods from ClassC1, ClassB1, or ClassD2.

It is also important to note that, in our example as in many other object-oriented designs, a specific object can be an instance of more than one class. For example, based on the way we have shown the class inheritance, any ClassD1 object is not only an instance of ClassD1, but is also an instance of ClassB2 as well as an instance of ClassA. In simpler terms, we can say that a ClassD1 object ***is*** also a ClassB2 object and ***is*** also a ClassA object.

You can best understand how class inheritance works by considering a concrete, but hypothetical example. Suppose that you were designing an object-oriented database to store information about vehicles. An object-oriented design of such a database might be shown as follows:

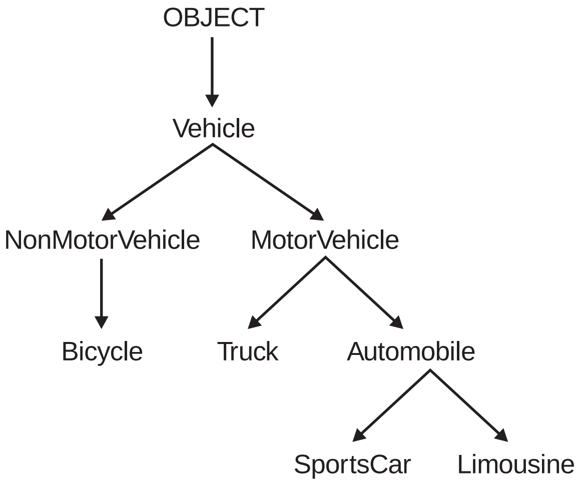


Figure 12. Object Class Inheritance

In our example, every object belonging to the Vehicle class also belongs to the OBJECT class. [1] The Vehicle class is divided into two more specific subclasses: the NonMotorVehicle and the MotorVehicle classes. Notice that the NonMotorVehicle class only contains one subclass: the Bicycle class. On the other hand, the MotorVehicle class contains two subclasses: the Truck class and the Automobile class. The Automobile class, in turn, contains several subclasses of its own; the only two shown are the SportsCar and the Limousine classes.

Assume that all of the classes shown are abstract except for the Bicycle class, the Truck class, and the subclasses of Automobile. This means that there is no OBJECT in our database that is not something more specific; for example, an OBJECT can be a Vehicle. Likewise, there are no Vehicle objects in our database that are not additionally classified as a specific type of Vehicle, and so on. Furthermore, there is no Automobile object that is not a specific type of Automobile: for example, a specific instance of the Automobile class must be either a SportsCar, a Limousine, or an instance of one of the other Automobile subclasses not shown. It is also important to note that, in our example, every object inherits from the generic OBJECT class. The OBJECT class is not only abstract, but it can be referred to as the *base class* of our database because all instances inherit from it.

Suppose we have stored objects in our database that represent specific vehicles, including Frank's sports car and Jackie's limousine. Frank's sports car is an

---

1. Assume that the database for this example may contain other subclasses of OBJECT in addition to Vehicle, as well as other subclasses for Automobile in addition to SportsCar and Limousine. These subclasses are not shown in Figure 12 to simplify our discussion.

instance of the SportsCar class. It ***is*** a SportsCar. We can also say that Frank's sports car ***is*** an Automobile, that it ***is*** a MotorVehicle, that it ***is*** a Vehicle, and that it ***is*** an OBJECT. After all, any sports car can be classified as an automobile, as well as a motor-powered vehicle, a generic vehicle, or even as a physical object. Similarly, Jackie's limousine is a Limousine, an Automobile, and so on.

The type of inheritance that has been described is also known as *single inheritance*. *Single inheritance* is a design characteristic of an object-oriented database in which each class inherits directly from only ***one*** immediate superclass. For example, even though SportsCar inherits from Automobile, which inherits from MotorVehicle and so on, SportsCar only inherits ***directly*** from ***one*** immediate superclass: Automobile. This is true for all the other classes in our vehicle database because it is designed using single inheritance.

You can best understand single inheritance by considering its alternative, which is *multiple inheritance*. *Multiple inheritance* is a design characteristic of an object-oriented database in which a given class can inherit directly from more than one immediate superclass. Our original vehicle database was not designed using multiple inheritance because no class inherited directly from more than one immediate superclass. Another vehicle database could implement multiple inheritance in the following way:

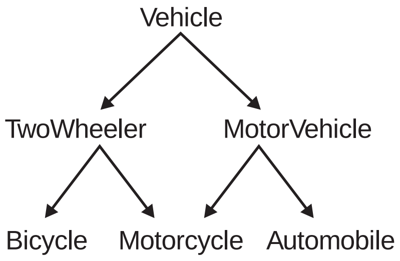


Figure 13. Multiple Inheritance Example

This is an example of multiple inheritance because Motorcycle inherits directly from both TwoWheeler and MotorVehicle. Note that multiple inheritance and single inheritance are mutually exclusive.

Collection Parts Stored in the TPF Database

In object-oriented design, objects that are unrelated to each other in terms of inheritance are often grouped together to represent other objects. This practice of grouping objects together for this purpose is commonly known as *object aggregation*. Object aggregation is a way in which objects of different classes relate to one another and has nothing to do with class inheritance. You can best understand object aggregation by considering how it can serve to design another database used to store vehicle information. Consider the following example, which shows how objects of the Automobile class relate to objects of various different classes even though none of these classes may be related in terms of inheritance. Even though the Automobile class defines a whole set of attributes that are inherited by objects belonging to this class, other objects might be needed in addition to these attributes to represent a real automobile. Some, but not all, of these objects are shown in the following figure:

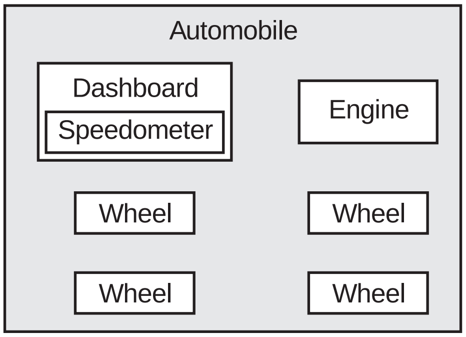


Figure 14. Object Aggregation

To represent an actual automobile in our database in greater detail, we use an Automobile object to store some generic attributes about the automobile as well as a whole series of other objects such as Wheel objects, an Engine object, a Dashboard object, and so on. All of these objects, including the Automobile object itself, are considered as parts of a greater whole that represents an actual automobile.

Source Code Definition of Objects

Object classes are defined in the TPFCS source code with the internally used CLASSC macro. A DSECT following each CLASSC statement defines the attributes inherited by all objects belonging to a particular class. Each field in the DSECT describes an attribute.

To understand how TPFCS defines class attributes and inheritance, consider the following example, which describes a subset of our original vehicle database.

```
OBJECT           CLASSC    ,
                 ATTRIBUTES TYPE=INSTANCE
ObjectHeader     DS  0CL16
ObjectId         DS  F            class ID of object
ObjectSeqCtr     DS  F            update sequence counter
ObjectLength     DS  F            length of this object
                 DS  F            reserved
                 ENDATTRIBUTES TYPE=INSTANCE
  ⋮
                 ENDCLASS

Vehicle          CLASSC SUPERCLASS=OBJECT

                 ATTRIBUTES TYPE=INSTANCE
Owner            DS  CL16         name of the Vehicle Owner
OwnerAddress     DS  CL16         address of  Vehicle Owner
RegisterRequired DS  CL1          'Y' == reg. required by law
*                                 'N' == reg. is optional
RegisterSatus    DS  CL1          'Y' == Yes, registered
*                                 'N' == No, not registered
VehicleIdNumber  DS  CL32         Vehicle Identification Number
*                                  or blanks if not registered
BrandName        DS  CL16         Brand Name or Manufacturer
ModelName        DS  CL16         Model Name or blanks
Year             DS  CL4          Year of manufacture or zero if
*                                 unknown
*
                 ENDATTRIBUTES TYPE=INSTANCE
  ⋮
                 ENDCLASS
```

Figure 15. Defining TPF Collection Support Source Code Instance Attributes (Part 1 of 2)

```
MotorVehicle CLASSC SUPERCLASS=Vehicle

                  ATTRIBUTES TYPE=INSTANCE
PowerSource       DS  CL16         Main Power Source
*                                  e.g. (UN)LEADEDGAS, GASOHOL, DIESEL
NumberOfWheels    DS  H            Number of Wheels
LicensePlate      DS  CL10         License Plate Number or blanks
                  ENDATTRIBUTES TYPE=INSTANCE
  ⋮
                  ENDCLASS

Automobile        CLASSC SUPERCLASS=MotorVehicle

                  ATTRIBUTES TYPE=INSTANCE
NumberOfDoors     DS  H            Number of Doors
NumberOfSeats     DS  H            Number of Seats
Radio             DS  C            'Y' if radio, 'N' if none
AirConditioner    DS  C            'Y' if a/c,   'N' if none
ExteriorColor     DS  CL10         Color of Exterior
                  ENDATTRIBUTES TYPE=INSTANCE
  ⋮
                  ENDCLASS

Limousine         CLASSC SUPERCLASS=Automobile

                  ATTRIBUTES TYPE=INSTANCE
Bar               DS  C            'Y' if bar,   'N' if none
Refrigerator      DS  C            'Y' if fridge,'N' if none
TelevisionBrand   DS  CL10         Brand of TV or blanks if none
                  ENDATTRIBUTES TYPE=INSTANCE
  ⋮
                  ENDCLASS
```

Figure 15. Defining TPF Collection Support Source Code Instance Attributes (Part 2 of 2)

Note: In addition to method attributes, TPFCS also defines class attributes and methods for each object class, which are not shown in Figure 15 on page 162. Class attributes are internal constants and are not contained in the objects themselves, nor are they written to the TPF database. The object methods are present only in the TPFCS load modules and also are ***not*** written to the TPF database in file copies of objects.[2]

The example of source code we have just cited describes the attributes of all objects of the Limousine, Automobile, MotorVehicle, Vehicle, and OBJECT classes. The SUPERCLASS parameter on each CLASSC macro establishes the inheritance relationship among these classes by dictating that a Limousine is an Automobile, which, in turn, is a MotorVehicle, and so on. Because of this inheritance relationship, any Limousine object would not only inherit the attributes defined for the Limousine class, but would inherit the attributes defined for the other classes as well.

It is important to realize that the inheritance relationship among the classes would still exist even if the CLASSC macro defining each class was located in a separate source module.

---

[2] In standard object-oriented terminology, *objects* refer to encapsulated data, which consists of both the attributes and methods. Because TPFCS only includes the data stored in instance attributes of an object when it is written to the TPF database, from this point on in our discussion, whenever we refer to *objects* we are referring exclusively to data.

File Representation of Objects

TPFCS is implemented using many of the object-oriented concepts previously described. All data in a TPFCS database is stored and maintained using objects. The data comprising an object consists of the attributes that the object inherits from its own class, preceded by all the attributes the object inherits from higher classes.

There are several similarities between the design of the previously discussed sample vehicle database and the design of the TPFCS database. Recall that, in our vehicle database, we have an abstract OBJECT class that is the base class; that is, all objects in the database inherit attributes and methods from the OBJECT class. TPFCS also implements an abstract class named OBJECT as the base class from which all objects stored in the database inherit.

Furthermore, just as in the vehicle database, the TPFCS object-oriented design uses single inheritance and, therefore, does ***not*** use multiple inheritance.

Suppose TPFCS created an object of ClassC, where ClassC objects inherit from higher classes as follows:

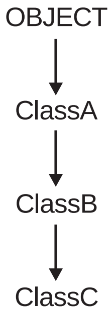


Figure 16. ClassC Inheritance Scheme

TPFCS would store a ClassC object in a pool record in the TPF database as follows:

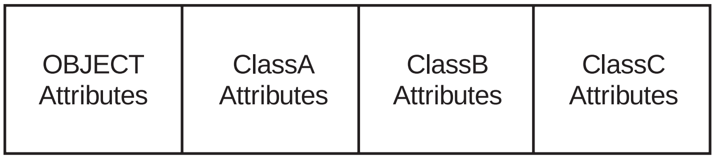


Figure 17. A ClassC Object

Because all of the objects in the TPFCS database design inherit from the base OBJECT class, the attributes of the OBJECT class always precede all other attributes contained in a TPFCS object. We refer to the attributes the object inherits from the OBJECT class as the *object header*. As was the case for the definition of our hypothetical vehicle database objects in Figure 15 on page 162, one of the attributes inherited from OBJECT is the class ID, which is an identifier in the object header that indicates the most detailed classification of the object. This identifier would always indicate the most immediate (and detailed) class to which the object belongs. Therefore, for a ClassC object, even though it also belongs to other classes (such as ClassA and ClassB), this attribute would identify the object as an instance of ClassC.

Keeping in mind that the attributes inherited from the OBJECT class constitute an object header, which, in turn, identifies or describes the object, the following is another way to depict the same ClassC object as stored in a 4-K pool record:

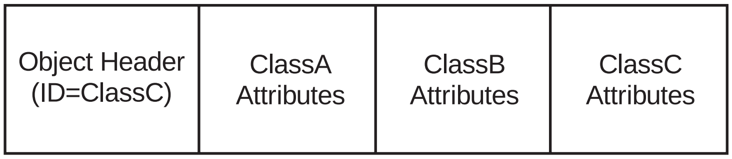


Figure 18. Another Way to Depict the Same ClassC Object

As shown in Figure 18, the object header would identify the object in question as an instance of ClassC.

TPFCS sometimes chooses to store separate objects in the same record in the TPF database. For example, there might be other objects immediately surrounding our ClassC object in the same record:

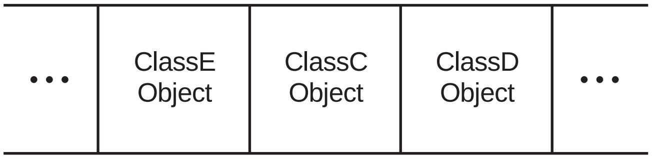


Figure 19. Separate Objects in the Same Record

In Figure 19, each object would contain its own unique header as well as its own unique set of attributes, including those it inherits from other classes. Furthermore, even though TPFCS can use the ClassC, ClassD, and ClassE objects to write information about the same collection in the TPF database, those objects would be ***unrelated*** to each other in terms of class inheritance. TPFCS could also choose to embed the same three objects in the *data area attribute* of yet another, larger object of ClassX to which they are also unrelated in terms of inheritance. A single record might then contain all these objects as follows:

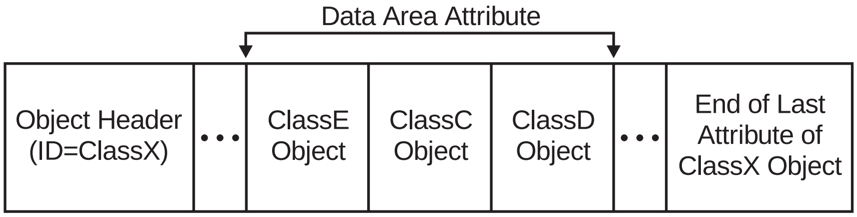


Figure 20. A ClassX Object. (Contains Other Objects Stored As Data in an Attribute)

A *data area attribute* is an attribute (field) within an object that is used to store generic data. Depending on how TPFCS implements the object, the generic data can be comprised of other objects, as in our example.

The easiest way to understand how TPFCS objects will appear in the database is to consider the vehicle objects of our hypothetical example described in Figure 15 on page 162. If a Limousine object were written in the TPF database in the same way as an object created by TPFCS, it would appear in a 4-K pool record as follows:

Object Header (ID=Limousine)	Vehicle Attributes	MotorVehicle Attributes	Automobile Attributes	Limousine Attributes

Figure 21. A Limousine Object

Similarly, other objects associated with the Limousine would be embedded in the data area attribute of a ClassX object as shown below:

AutoProxy Object	Dashboard Object	Engine Object	Wheel Object	Wheel Object	Wheel Object	Wheel Object

Figure 22. Data Area Attribute of a ClassX Object in the Vehicle Database

The ClassX object shown would appear in a separate 4-K pool record from the one containing the Limousine object. The AutoProxy object contained in its data area would contain an attribute where the file address of the Limousine object record is stored. In this way, the ClassX object is used to store or locate all the objects used to represent a real limousine.

OBJECT Class

As previously mentioned, every class in TPFCS has the OBJECT class as its base class; that is, all classes inherit from the OBJECT class. For this reason, all objects are said to have an object header, which consists of the attributes of OBJECT.

Object Header

The object header contains the following data in the order presented:

Object ID (class ID)
A 4-byte field containing a hexadecimal value known as the *class ID* of the object. The class ID stored in this field for a given object represents its most detailed classification. This field is the same attribute discussed in our previous abstract examples and identified an object as being either a Limousine or a ClassC object, and so on. The class IDs of all of the objects used by TPFCS are defined by CLASSID macro statements in the ITO2 DSECT.

Update sequence counter
A 4-byte sequence counter field used by some application programming interfaces (APIs).

Object length
A 4-byte field containing the length of the object when it is brought into memory.

The value stored in this field will include the lengths of all attributes inherited from higher classes, including the length of the object header itself.

spare
An additional 4 bytes is reserved for future use by IBM.

Use of Pool Records

The objects used internally by TPFCS to represent collections and their elements are stored in TPF long-term 4-K pool records. TPFCS will use FARF3, FARF4, or FARF5 4-byte file addresses or FARF6 8-byte file addresses. The type of address that is used depends on two things: whether the TPF system has been set to allow FARF6 file address dispensing or when the collection was created. Once the TPF system is set to allow FARF6 file address dispensing, all new collections will be created using 8-byte file addresses. To set the TPF system to allow 8-byte file addressing, enter **ZMODE 6** to set on the SB8BFAD SYSTC switch (see *TPF Operations* for more information about the ZMODE command). To begin using 8-byte file addressing, you must use the record ID attribute table (RIAT) characteristics associated with the TPFCS record IDs (4-byte collections use RIAT RTP=0 and 8-byte collections use RIAT RTP=1). The following section describes how TPFCS logical objects are physically represented on DASD.

TPFCS Primary and Shadow Records

When a collection is created using the shadow option, TPFCS maintains a duplicate copy of each of the records used to represent the collection in the TPF database. This duplication feature that TPFCS provides is known as *shadowing*, and all collections that are duplicated in this way are said to be *shadowed*. TPFCS provides shadowing independent of TPF duplication of files. This means that if the pool records used to represent a shadowed collection are duplicated by the TPF system, there are four copies of each record.

The first copy of each record TPFCS uses to represent a collection is known as a *primary record*. If a collection is not shadowed, TPFCS uses only primary records to represent the collection. If the collection is shadowed, TPFCS maintains a copy of each primary record used to represent the collection. Each copy is known as a *shadow record* or simply, a *shadow*. Any time it needs to read one of these records, TPFCS can randomly read the shadow of that record rather than the primary copy of that record to maintain database integrity.

See *TPF Operations* for more information about the ZOODB DEFINE command with the SHADOW parameter.

TPFCS Record Header

Figure 23 describes the record header that TPFCS uses to file its objects in 4-K pool records using the 4-byte file address format:

```
IDSDGP   DSECT
DGPTPF_HDR DS  0XL24             TPF record header
DGPRID     DS  XL2               record ID
DGPRCC     DS  X                 record code check
DGPCTL     DS  X                 record format flag
DGPFORMAT1 EQU X'01'             1 - current format
*                                other value reserved for future use.
DGPPRGNM   DS  XL4               program name
DGPFWD     DS  XL4               reserved
DGPBKD     DS  XL4               reserved
DGPPRIME   DS  XL4               file address of primary record
DGPSHADOW  DS  XL4               file address of shadow or zero
DGPTPF_HDR_SIZE EQU *-DGPTPF_HDR
```

Figure 23. DSECT Showing the 4-Byte Format Record Header

Figure 24 describes the record header that TPFCS uses to file its objects in 4-K pool records using 8-byte file address format:

```
IF6DP    DSECT
I6PTPF_HDR DS  0XL48              TPF record header
I6PRID     DS  XL2                record ID
I6PRCC     DS  X                  record code check
I6PCTL     DS  X                  record format flag
I6PFORMAT0 EQU X'00'              0 - current format
I6PFORMAT1 EQU X'01'              1 - current format (alternative)
I6PFORMAT2 EQU X'02'              2 - expanded (FARF6) format
I6PPRGNM   DS  XL4                program name
           DS  XL4                Reserved for future IBM use
           DS  XL4                Reserved for future IBM use
I6PFWD     DS  XL8                file address of next in chain
I6PBKD     DS  XL8                reserved
I6PPRIME   DS  XL8                file address of prime
I6PSHADOW  DS  XL8                file address of shadow or 0
I6PTPF_HDR_SIZE EQU *-I6PTPF_HDR
```

Figure 24. DSECT Showing the 8-Byte Format Record Header

It is important to note how prime and shadow fields are used. Consider the following example using the 4-byte file address format in which the pool record at file address `180BF6A6` is used by TPFCS:

```
User:   ZDFIL 180BF6A6 000.1F

System: CSMP0097I 16.32.50 CPU-B SS-BSS  SSU-HPN  IS-01
        DFIL0010I 16.32.50 BEGIN DISPLAY OF FILE ADDRESS 180BF6A6
         00000000- FC160100 C3D1F0F0 00000000 00000000 ....CJ00 ........
         00000010- 180BF6A6 180BF6A7 0000004C 00000000 ..6w..6x ...<....
        END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

Figure 25. TPF Collection Support Primary Record Header Using 4-Byte File Addresses

In Figure 25, the primary record points to itself because, at the displacement corresponding to its DGPPRIME field, the file address of the primary record (`180BF6A6`) is stored. The primary record also points to the shadow using DGPSHADOW, which contains the file address of the shadow record (`180BF6A7`). Likewise, the shadow record, whose contents match the primary record exactly, points back to the primary record using the DGPPRIME field as well as to itself through the DGPSHADOW field.

```
User:   ZDFIL 180BF6A7 000.1F

System: CSMP0097I 16.32.50 CPU-B SS-BSS  SSU-HPN  IS-01
        DFIL0010I 16.32.50 BEGIN DISPLAY OF FILE ADDRESS 180BF6A7
         00000000- FC160100 C3D1F0F0 00000000 00000000 ....CJ00 ........
         00000010- 180BF6A6 180BF6A7 0000004C 00000000 ..6w..6x ...<....
        END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

Figure 26. TPFCS Shadow Record Header

If shadowing were not active for the record filed at file address `180BF6A6`, our example might be as follows:

```
User:   ZDFIL 180BF6A6 000.1F

System: CSMP0097I 16.32.50 CPU-B SS-BSS  SSU-HPN  IS-01
        DFIL0010I 16.32.50 BEGIN DISPLAY OF FILE ADDRESS 180BF6A6
         00000000- FC160100 C3D1F0F0 800814CE 00000000 ....CJ00 ........
         00000010- 180BF6A6 00000000 0000004C 00000000 ..6w..6x ...<....
        END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

Figure 27. Primary Record Header with No Shadow

We know that the record is not shadowed because the displacement that corresponds to the DGPSHADOW field contains zero.

TPFCS Record Trailer

In addition to having a header, the records in which TPFCS files its objects contain a trailer. The following DSECT outlines the contents of this trailer:

```
              DSECT
DGPPG_TRAILER DS  0XL14           OODB PAGE TRAILER
DGPPG_RRN     DS  XL4             RRN of this record
DGPPG_ID      DS  XL8             File address of the owning control record
DGPPG_RESV    DS  X               reserved for future use
DGPPG_FFLG    DS  X               TPF format flag
```

Figure 28. Contents of the TPFCS Record Trailer

For more information about a relative record number (RRN), see “Extended Structures (StructureDasd Class)” on page 181. For more information about the owning control record, see “Collection Control Record” on page 170.

Packaging in DATXPAGE Envelopes

TPFCS uses object aggregation to represent actual collections (see “Collection Parts Stored in the TPF Database” on page 160). Apart from the many data elements that can be contained in a collection, TPFCS groups more than one object together to represent the collection itself. Objects such as these are referred to as *collection part objects*, which are also known as *collection parts* or *part objects*. TPFCS uses part objects to store control information about collections as well as to hold or point to the data elements that are contained in those collections. To be a collection part, an object must inherit from the ObjectPart class as defined in the TPFCS source code. Do not let the name of this class mislead you: a collection part, while a member of the ObjectPart class, is itself an entire object because the ObjectPart class inherits from the OBJECT class. [3]

The group of internal part objects that represent a collection are packaged into a larger object, which is an instance of the DATXPAGE class, and will be referred to as a *DATXPAGE envelope*. A DATXPAGE envelope appears in a pool record after the TPF record header as follows.

Note: The DATXPAGE envelope will contain embedded objects.

---

[3] Recall our discussion in “Object-Oriented Concepts” on page 157 in which a SportsCar is an Automobile, as well as a MotorVehicle, as well as a Vehicle, as well as an OBJECT. Similarly, any given ObjectPart is also an OBJECT.

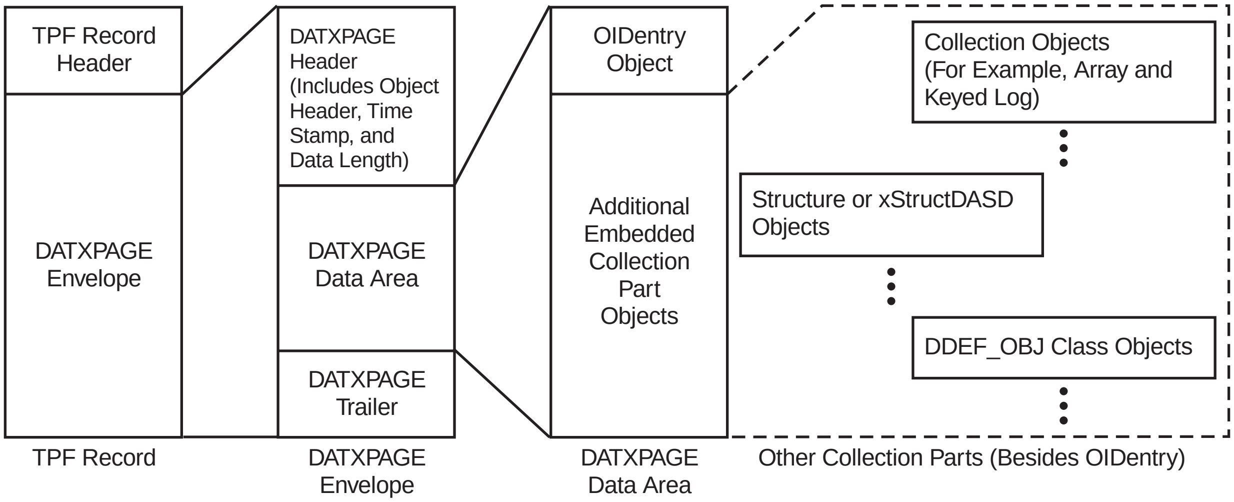

TPF Record Header
DATXPAGE Envelope
TPF Record
DATXPAGE Header (Includes Object Header, Time Stamp, and Data Length)
DATXPAGE Data Area
DATXPAGE Trailer
DATXPAGE Envelope
OIDentry Object
Additional Embedded Collection Part Objects
DATXPAGE Data Area
Collection Objects (For Example, Array and Keyed Log)
Structure or xStructDASD Objects
DDEF_OBJ Class Objects
Other Collection Parts (Besides OIDentry)

Figure 29. DATXPAGE Envelope Format

Figure 29 shows that:

- The DATXPAGE envelope follows the TPF record header.
- The *object header* in the DATXPAGE header is the same standard header that all objects have (see “Object Header” on page 166), which includes an object length field. The value for the length stored in this field is the size of the entire DATXPAGE envelope.
- The *time-stamp* attribute in the DATXPAGE header is an 8-byte field containing the time that TPFCS last filed that record.
- The *data length* attribute in the DATXPAGE header is a 4-byte field containing the hexadecimal number of bytes of the data area in use.
- The *data area* contains the objects (all of which are collection parts) that TPFCS has embedded or packaged in that DATXPAGE envelope. The only time the data in this area is not an actual object is when an overflow condition results. See “How an Object Can Overflow into Additional Records” on page 210 for more information about this condition.
- The DATXPAGE trailer is the same as the TPFCS record trailer. See Figure 28 on page 169 to determine its contents.

The DATXPAGE envelope and all the collection parts contained in it are filed in the TPF database in a record known as the *control record*.

Collection Control Record

Every persistent collection has a control record associated with it. To an application programmer, the unique persistent identifier (PID) associated with a given collection serves as a token that applications must supply to TPFCS to perform any operation on the collection after it is created. The PID represents an anchor to which an associated collection and all its data are tied. From the perspective of a database administrator, the *control record* serves as the anchor to which all of the parts that represent an associated collection (including its data) are chained.

The control record contains a DATXPAGE envelope in which all of the collection part objects used to represent a collection are stored. The following example shows the contents of a control record for an array collection that is not shadowed:

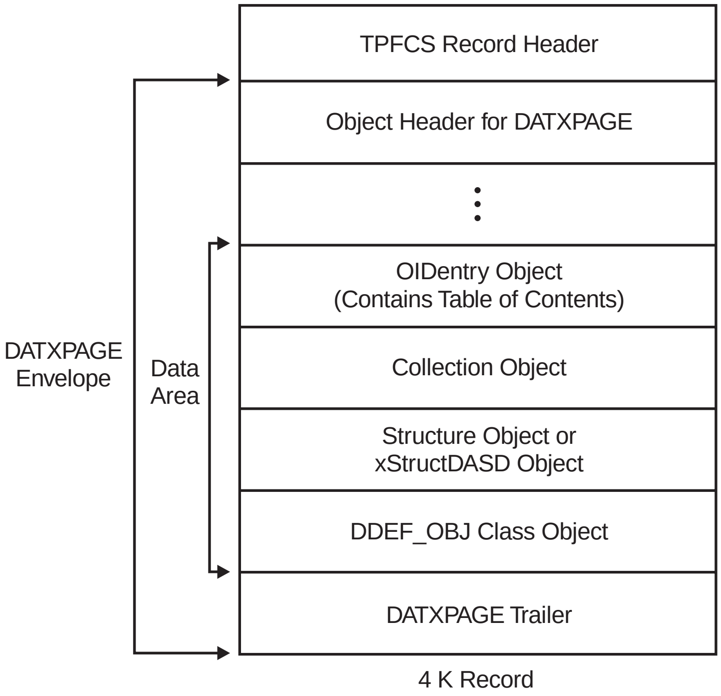


Figure 30. Example of a Collection Control Record

The following objects are all packaged in the DATXPAGE envelope filed in the control record:

- OIDentry object:

  An OIDentry object is the first collection part packaged in the DATXPAGE envelope. Its purpose is to serve as a table of contents by identifying how many other collection parts follow in the envelope and where each part begins (relative to the beginning of the OIDentry object itself). The OIDentry object contains a table of contents attribute with as many entries as there are additional objects in the DATXPAGE envelope. Each entry contains:
  - A displacement relative to the start of the OIDentry itself where another object has been stored
  - The length of the object stored at that location.
- Collection object:

  This varies according to collection type. For example, if the collection is an array, its collection object is an ARRAY object. The collection object contains information that is unique to the type of collection. For example, for arrays, one of the attributes stored in the ARRAY object is the next index to be used.
- Structure object or xStructDASD object:

  For compact-resident collections, the structure object follows (see “Collection Residency” on page 173 for more information about residency). This is the object that contains control information as well as the actual data elements stored in those collections.

  For extended-resident collections, the DATXPAGE envelope contains an xStructDASD object instead of a structure object because the structure object is too large. The xStructDASD object, in turn, contains a pointer to the structure object. This structure object will contain control information used to locate the data elements elsewhere in the TPF database, rather than containing the data elements themselves.

Note: For compact-resident collections, the *structure record* and *control record* are one and the same. As a result, the file address for the structure record for compact-resident collections should be the same as the file address contained in the collection PID.

- DDEF_OBJ object:

  This is an object that contains information about the data definition used to store the collection.

TPFCS stores some types of collections using more than one structure. For example, a keyed log collection can be stored using the same structure object used for regular (nonkeyed) logs to contain the data, and another structure that is the same structure object as a key set collection to maintain the key associated with each entry there. When this occurs, TPFCS will use a separate collection object as well as a separate DDEF_OBJ object for each structure object. In our example, TPFCS would implement the keyed log collection using both a log collection as well as a key set collection. There would still be just one PID assigned to the keyed log. The contents of the control record associated with the PID would be as follows:

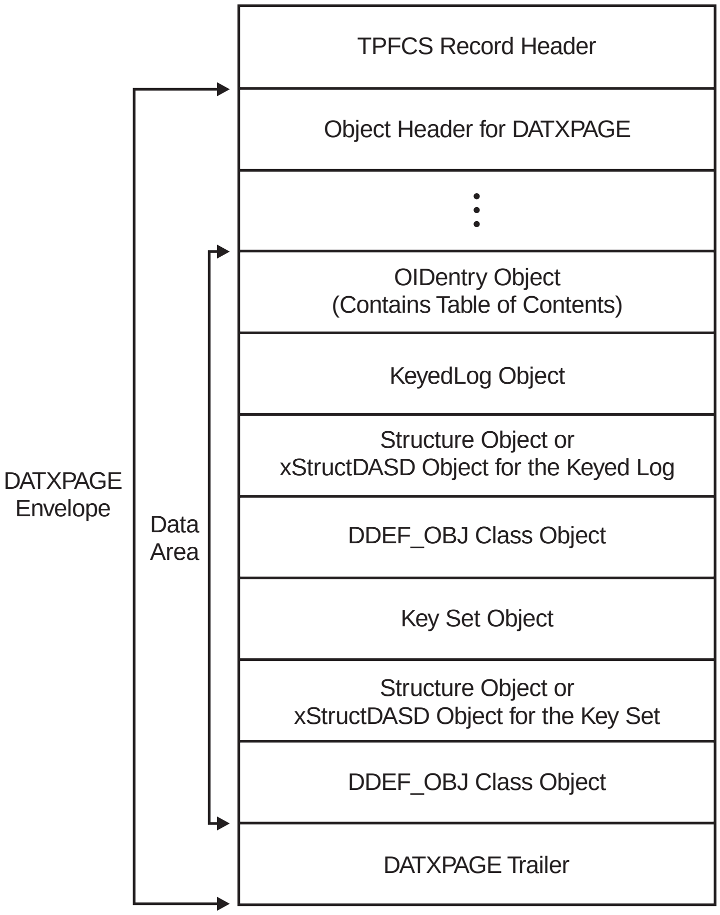


Figure 31. Conceptual View of a Control Record for a Keyed Log Collection

To make it easier, all other discussions in this publication assume a single structure for a collection unless otherwise noted. Furthermore, our discussion of both the DATXPAGE envelope and the control record has been based upon a logical representation. In reality, it is possible that all of the objects which TPFCS intends to package in a single DATXPAGE envelope and corresponding control record will not fit in a single 4-K pool record on DASD. See “How an Object Can Overflow into

Additional Records” on page 210 for detailed information about how a logical control record and the objects it contains are split across multiple physical records.

USERdata Object

TPFCS often stores the data elements in a collection in USERdata objects. USERdata objects are embedded in other objects that TPFCS uses in its internal representation of the collection.

A USERdata object only inherits from the OBJECT class. This means that it only contains a standard object header followed immediately by the actual user data, which is the data element itself.

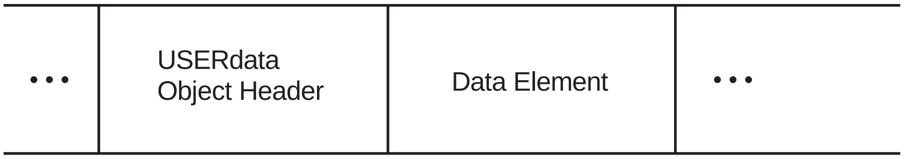


Figure 32. USERdata Object

Collection Residency

TPFCS classifies collections by a characteristic known as *residency*, which determines what the layout of the data is in the internal objects that comprise a collection. The two types of residency are *compact* and *extended*. For compact-resident collections, the data elements are embedded in the structure object itself. For extended-resident collections, the structure object only contains control information about the collection structure and the data elements are embedded in data records, which are chained to the structure object.

Note: Because the data elements are embedded in compact structures, it is more effective to implement collections with fewer elements using compact structures. Likewise, it is more effective to implement collections with a very large number of elements using extended structures because of the overhead that is necessary to sort and retrieve those elements.

Every collection can be represented by a compact or extended structure. When a collection is first created it is compact-resident. TPFCS changes the internal representation of the collection to be extended-resident only when the collection has exceeded a predetermined size as more and more elements are added to it. Once TPFCS has made a collection extended-resident, the collection will remain that way even if the number of elements in the collection is significantly reduced.

Note: Collections are not transformed back and forth from extended-resident to compact-resident to avoid overhead. There would be a performance impact for this transformation as a collection continued to increase and decrease in size as many elements were added to it and many were deleted from it.

Compact Structures (StructureMem Class)

All compact structure objects inherit from the StructureMem class. This section describes the specific types of compact structure objects provided by TPFCS:

- MemFLAT
- MemHash

- MemKey
- MemList.

Every compact structure will be an instance of one of these object classes. You will notice that all of these classes are similar in that they each have a data area attribute, but they are different because each has their own specific attributes used to manage the data.

MemFLAT

Of all the structure objects belonging to the StructureMem class, the MemFLAT object is the most simple. The *MemFLAT structure object* contains a data area attribute where TPFCS stores the collection data. The collection data is stored in the exact order and format the applications using the collection have specified. Each time an application requests data to be stored or retrieved from the collection, TPFCS uses a relative byte position as the displacement into the data area. The applications that store, retrieve, or update data in MemFLAT collections specify this relative byte position either directly or as an index.

Important Attributes

The following are important MemFLAT attributes that precede the data area:

MemFLAT_I_FAL
A 4-byte field containing the number of bytes in the data area available to add new data.

MemFLAT_I_LGH
A 4-byte field containing the length of the data area where the collection data is stored.

MemFLAT_I_NAB
A 4-byte field containing the displacement into the data area where new data can be added.

Graphical Representation

To understand how the *MemFLAT object* data area will appear on file, it is helpful to understand how TPFCS sets it up in memory. The following figure is a conceptual representation of how TPFCS lays out a MemFLAT object data area in memory:

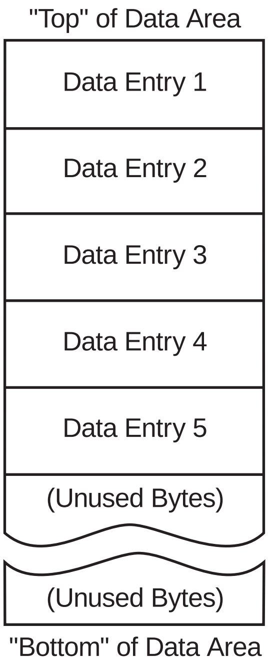


Figure 33. Example of the Data Area of a MemFLAT Object

Data Format

Each data entry contains the actual user data.

MemHash

TPFCS uses the MemHash object to employ a hashing algorithm to store and retrieve data in collections. The hashing algorithm makes use of an *input value*, which is extracted from the input that is provided by application programs requesting to store and retrieve data from those collections as well as a *hash value*, which is produced by the algorithm.

Important Attributes

The following attributes of the MemHash object are most important:

MemHash_I_DataArea
The data area where the collection elements and their hash values are stored. This data area is a contiguous list of data entries that contain the elements of the collection and their hash values.

MemHash_I_FAL
A 4-byte field containing the number of bytes that are available to store additional data and control information.

MemHash_I_LGH
A 4-byte field containing the length of the data area.

MemHash_I_MaxEntryLgh
A 2-byte field containing the maximum data entry length.

MemHash_I_NAB
A 4-byte field containing the number of bytes that are in use to store both data and control information.

MemHash_I_NumEntries
A 2-byte field containing the number of entries or elements stored in the collection.

MemHash_I_ValueLgh
A 2-byte field containing the length of the input value for each data element on which TPFCS will perform a hashing algorithm to locate that element.

Graphical Representation

TPFCS inserts entries into the data area as applications insert elements in the corresponding collection. There is no sorting of elements and there is no chaining of hash synonyms. For example, the data area of a MemHash structure object could contain the following:

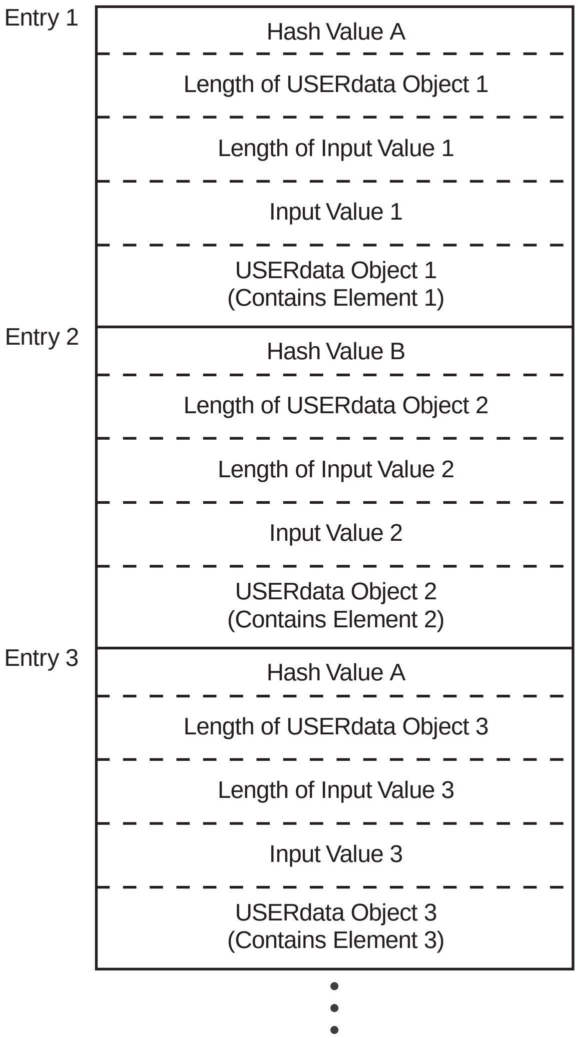


Figure 34. Example of the Data Area of a MemHash Object

To retrieve an element in the collection, TPFCS searches the data area sequentially for all occurrences of its hash value until the element input value is found. In Figure 34, if TPFCS needed to process a request to retrieve Element 3, it would perform the MemHash hash algorithm on Element 3 using Input Value 3, which would either be a portion of Element 3 specified by the requesting application or

Element 3 itself. [4] The result of the hash operation would be Hash Value A. TPFCS would then search the MemHash data area for a match on Hash Value A that also contained a matching input value. Element 1 would be found but would be bypassed because Input Value 1 does not equal Input Value 3. TPFCS would continue its search and retrieve Element 3 because its entry in the data area contains both a matching hash value as well as a matching input value.

Data Format

Each entry in the data area consists of the following:

- A 4-byte hash value that TPFCS produced to locate a corresponding data element the next time it is accessed.
- A 2-byte field containing the length of the USERdata object where the data element is stored.
- A 2-byte field containing the length of the input value that was used to produce the hash value for the data element.
- The input value TPFCS used to produce the hash value for the data element. For some collections such as bags and sets, the input value is a replica of the data element itself.
- A USERdata object containing the data element.

MemKey

The MemKey object provides the fastest way to retrieve and store data elements in small collections which are sorted on key. It is also used to store data for other keyed collections which are not sorted.

Important Attributes

The following attributes of a MemKey object are most important:

MemKey_I_DataArea
A data area of varying length where the keys, their data elements, and locators that determine the order of those elements in the collection are stored.

MemKey_I_FAL
A 4-byte field containing the number of bytes remaining in the MemKey data area (**MemKey_I_DataArea**).

MemKey_I_FirstDeadSlot
A 4-byte field that indicates the displacement to the first inactive entry in the data area. When there are no inactive entries, TPFCS sets this attribute to X'FFFFFFFF'.

An inactive entry is an entry in the MemKey data area that corresponds to an element in the associated collection that an application program has deleted.

MemKey_I_KeyLgh
A 2-byte field that contains the length of a key used for a data element that TPFCS will store in this MemKey object. The length of every key is the same.

MemKey_I_LGH
A 4-byte field containing the length of the data area when TPFCS reads the MemKey object into memory. This length will differ from the actual length of the data area on file because TPFCS will strip the data area of its unused bytes before filing the MemKey object to the TPF database.

---

[4] Whether the input value is part of, or the same as, the collection element depends on the type of collection represented by the MemHash structure object.

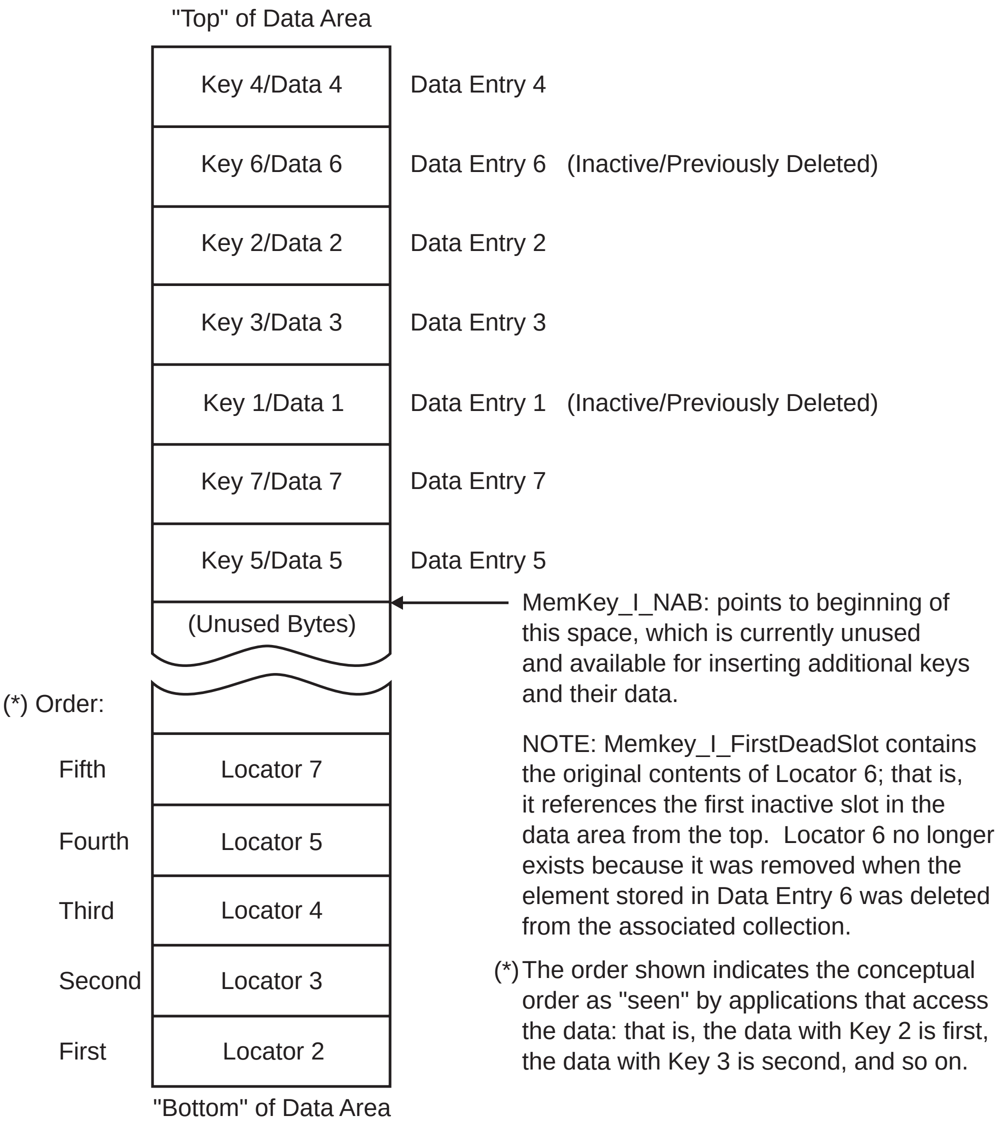
MemKey_I_MaxEntryLgh
A 4-byte field that contains the maximum size of a data element that TPFCS will insert into the MemKey data area.

MemKey_I_NAB
A 4-byte field containing the displacement in **MemKey_I_DataArea** where TPFCS will insert the next key and data element that is added to the collection.

MemKey_I_NumEntries
Contains the number of active data entries that the MemKey object currently holds. Each data entry is composed of a key as well as the data element that corresponds to that key in the collection.

Graphical Representation

To understand how the MemKey object data area will appear on file, it is helpful to understand how TPFCS sets it up in memory. The following figure is a conceptual representation of how TPFCS lays out a data area of MemKey in memory:



NOTE: Memkey_I_FirstDeadSlot contains the original contents of Locator 6; that is, it references the first inactive slot in the data area from the top. Locator 6 no longer exists because it was removed when the element stored in Data Entry 6 was deleted from the associated collection.

(*) The order shown indicates the conceptual order as "seen" by applications that access the data: that is, the data with Key 2 is first, the data with Key 3 is second, and so on.

Figure 35. Data Area of MemKey in Memory

The conceptual order of the data entries stored in a MemKey object differs greatly from the physical order in which TPFCS stores them. This conceptual order is determined by the order of the *locators*, which are displacement fields that TPFCS keeps at the bottom of the data area. TPFCS sorts the locators based on the keys of the data entries to which they point rather than sorting those entries themselves. When an application wants to delete an element from a collection that is represented by a MemKey object, TPFCS usually deletes the locator to the corresponding data entry that holds that element rather than deleting the entry itself. TPFCS only stores elements with unique keys in a MemKey object. As a result, if there appear to be entries with duplicate keys in a data area of a MemKey object, it is most likely that all but one of those entries are remnants of data elements that have been deleted from the associated collection. Even though TPFCS files MemKey objects whose data areas contain inactive entries, TPFCS always strips the data area of any gap of unused bytes between the last data entry and the start of the locators before filing a MemKey object. Therefore, when you display a record that contains a MemKey object, you will find its locators immediately following the last entry in the data area.

Data Format

The format of each data entry is as follows:

- The first 2 bytes contain the length of the entire entry. This length equals the length of the key, plus the length of the USERdata object that holds the element associated with that key, plus 2 (for the length field itself).
- The key associated with the data element. If the MemKey object holds a collection for which TPFCS allows duplicate keys, TPFCS adds a time stamp to this key to make it unique.
- A USERdata object that contains the actual data element.

Each locator is 4 bytes long and contains the following:

- The first 2 bytes contain the length of the data entry associated with that locator.
- The next 2 bytes contain the displacement from the start of the data area where the associated data entry begins.

Note: The format of the MemKey object when written to DASD differs from its format in memory because TPFCS condenses the object by removing all unused bytes from the data area. See “How Some Objects Are Condensed to Save Space” on page 207 for more information.

MemList

The MemList object is similar to the MemKey object because data elements stored in a MemList are sorted using locators. The main differences are that there are no keys associated with each element stored in a MemList object and there is no inherent order to those data elements. The order of the data elements for MemList structure collections is determined solely by the applications that create and populate them with elements.

Important Attributes

The following attributes of a Memlist object are most important:

MemList_I_DataArea
A data area of varying length where the data elements and locators that determine the order of those elements in the collection are stored.

MemList_I_DataLgh
A 4-byte field containing the length of the data area where elements are stored.

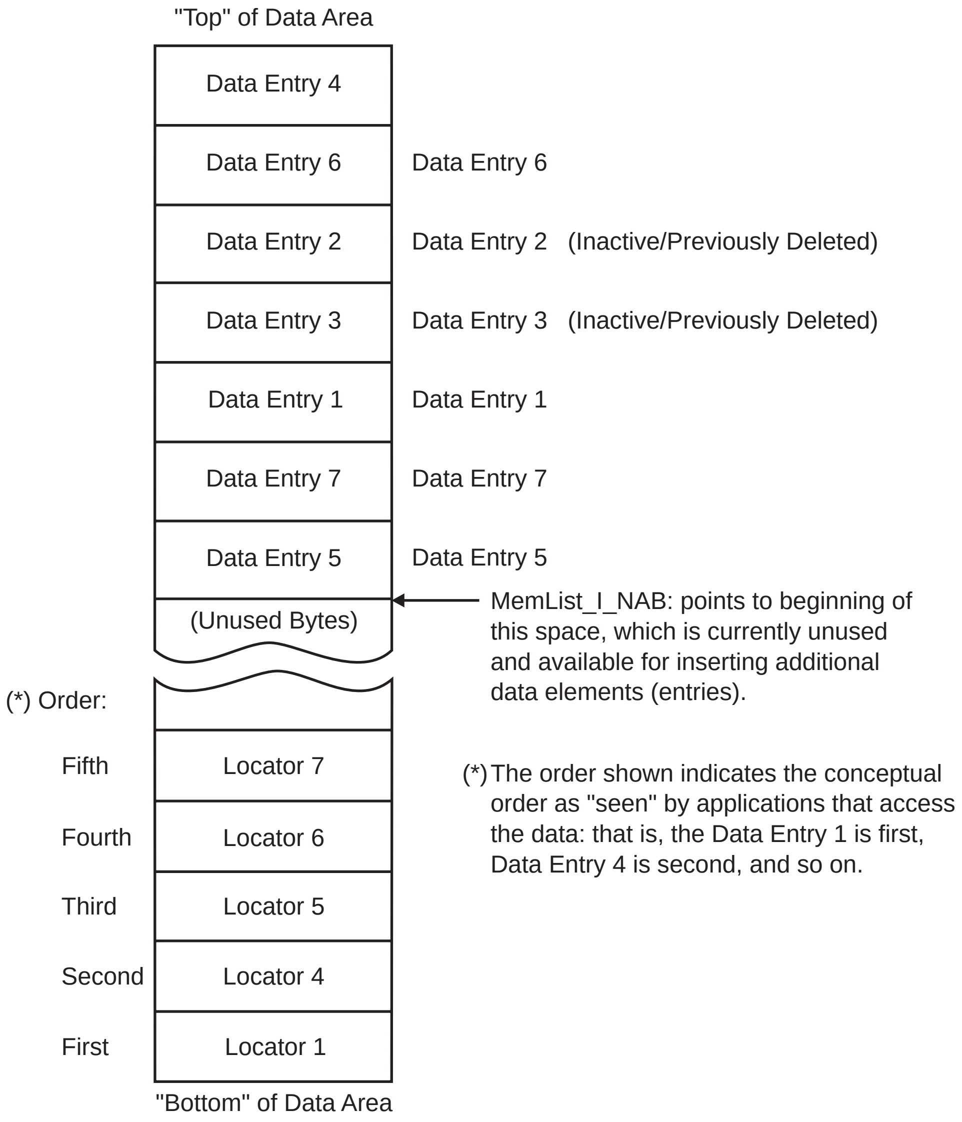
MemList_I_FAL
A 4-byte field containing the number of bytes available in the unused portion of the data area where the next element will be inserted.

MemList_I_NAB
A 4-byte field containing the displacement from the beginning of the data area where the next element will be inserted.

MemList_I_NumEntries
A 4-byte field containing the number of active entries (elements) in the collection.

Graphical Representation

To understand how the MemList object data area appears on file, it is helpful to understand how TPFCS sets it up in memory. The following figure is a conceptual representation of how TPFCS arranges a MemList data area in memory:



(*) The order shown indicates the conceptual order as "seen" by applications that access the data: that is, the Data Entry 1 is first, Data Entry 4 is second, and so on.

Figure 36. Example of a MemList Data Area in Memory

The conceptual order of the data entries stored in a MemList object differs greatly from the physical order in which TPFCS stores them. This conceptual order is determined by the order of the locators, which are displacement fields that TPFCS keeps at the bottom of the data area. TPFCS sorts the locators based on the order of the entries to which they point as indicated by the application rather than sorting those entries themselves. When an application wants to delete an element from a collection that is represented by a MemList object, TPFCS usually just deletes the locator to the corresponding data entry that holds that element rather than deleting the entry itself. Even though TPFCS files MemList objects whose data areas contain inactive entries, TPFCS will always strip the data area of any gap of unused bytes between the last data entry and the start of the locators. Therefore, when you display a record that contains a MemList object, you will find its locators immediately following the last entry in the data area.

Data Format

The format of each data entry is as follows:

- The first 2 bytes contain the length of the entire entry. This length equals the length of the USERdata object that holds the element stored in that entry plus 2 bytes (for the length field itself).
- A USERdata object that contains the actual data element.

Each locator is 4 bytes long and contains the following:

- The first 2 bytes contain the length of the data entry associated with that locator.
- The next 2 bytes contain the displacement from the start of the data area where the associated data entry begins.

Note: The format of the MemList object when written to DASD differs from its format in memory because TPFCS condenses the object by removing all unused bytes from the data area. See “How Some Objects Are Condensed to Save Space” on page 207 for more information.

Extended Structures (StructureDasd Class)

All extended structures inherit from the StructureDasd class. This section describes the following specific types of extended-structure objects provided by TPFCS as they apply to collections created with 4-byte file address format:

- DASDINDEX
- DASDFLAT
- DASDHASH
- DASDLIST.

Note: The architectural detail for collections created with 8-byte file address format is the same, except the object names will be different.

Record Types

Extended structures make use of the following:

- **Data records**

  TPFCS stores and retrieves data elements for extended-resident collections using *data records*. Data records are pool records where the actual data elements for the collection are stored. A data record often contains more than one data element for a given collection.
- **Key records (index records)**

  For collections with keys, TPFCS stores and retrieves data elements for extended-resident collections using *key* (*index*) *records* as well. Key records are pool records that are arranged in a tree-like structure by which TPFCS sorts the keys and locates the data record containing the data element that corresponds to each key. The root of the tree is chained to the structure object.

- **Directory entries** and **directory records**

  TPFCS also uses directory entries to represent extended-resident collections. Each directory entry that is in use for an extended-resident collection contains the file address of a key (index) or data record as well as its shadow if one exists. TPFCS uses the directory entries for a given collection to locate its key and data records by their *relative record number* (RRN). Some directory entries are embedded in the extended structure object (StructureDasd) itself, whereas others are kept in pool records, which are called *directory records*. Just as with key records, TPFCS chains the directory records for a collection in a tree-like structure whose root is chained to the structure object.

  Note: When a directory entry is not in use, it either contains zero or indicates the RRN for the next directory entry that is available.

A directory entry is not an object itself, but rather an attribute or field in the object in which it is contained. If you read the TPFCS source code, you must not confuse the directory entry with the DIRECTORY object, which is just one of the objects that TPFCS uses to store directory entries. A DIRECTORY object itself ***contains*** directory entries and is filed in a directory record.

A directory entry can have either of two formats and always contains a bit flag that identifies which format describes that entry. TPFCS uses directory entries in the first format to locate records on file.

In the first format, the following fields are contained in directory entries:

primary file address
A 4-byte field that contains the file address of a primary record used by TPFCS.

shadow file address
A 4-byte field that contains the file address of the shadow copy of the primary record or zero when a shadow does not exist.

The second format for a directory entry is used to store a next available RRN that TPFCS will use to represent a collection on DASD.

Note: For more information about the exact contents of a directory entry in either format, see the IDSDIRE DSECT in the ITO2 DSECT.

Relative Record Numbers

Each data record and key record (when present) belonging to an extended-resident collection is associated with an RRN. A key or data record relative record number describes the order in which that record occurs in the abstract flat file representation of the collection data and associated control information. You can best understand what is meant by a relative record number, as well as gain a better understanding of extended-resident collections, by considering the following abstract model. Even though it is not the case, you can think of all of the data of a given extended collection, as well as associated control information used to manage or sort through that data, as being stored in one huge continuous flat file. This flat file itself is partitioned into 4-K records:

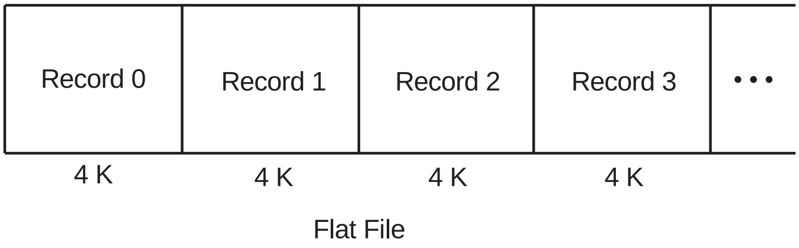


Figure 37. Abstract View of How Data Is Stored for Extended-Resident Collections

Each of the 4-K records used to contain or manage the data is either a data record or (for keyed collections) a key record. The first record in this file is assigned RRN 0, the next is assigned RRN 1, and so on.

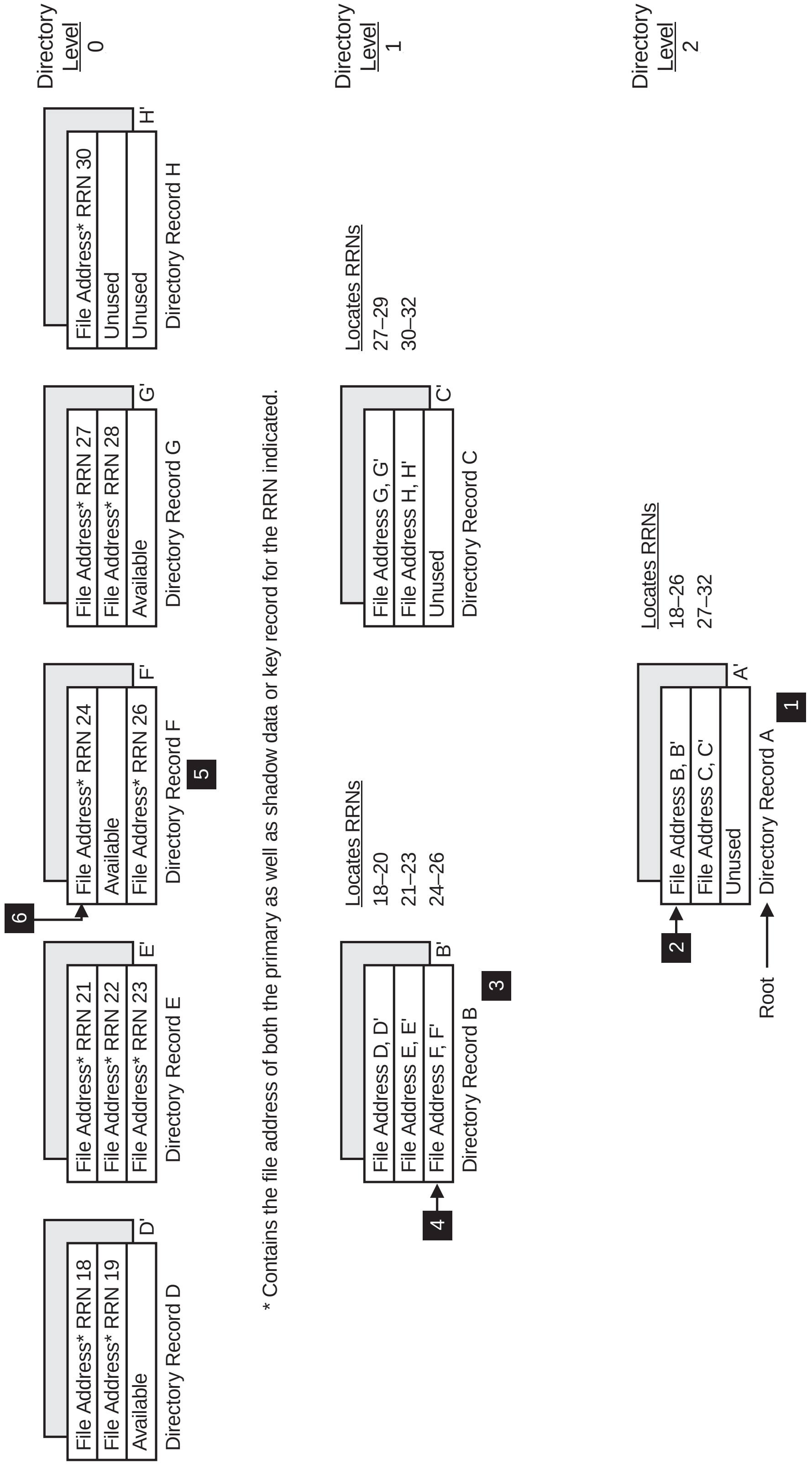
In reality, the flat file of this abstract representation does not exist. The 4-K records comprising the file are scattered across the TPF database in pool records. TPFCS uses directory entries as well as directory records to locate and access each of these records according to its relative record number.

Locating Records in a StructureDasd Object

As mentioned earlier, TPFCS uses directory entries and directory records to locate collection data and key records by their relative record number. Directory entries 0–17 are embedded in the collection structure object and TPFCS uses them to locate RRNs 0–17 directly: that is, when one of these RRNs is in use for a collection, the corresponding directory entry of the RRN will contain its file address (for both the primary record and the shadow when present). The directory records are used to locate RRNs 18 and higher. Logically, the directory records are arranged in a tree-like structure with the root of the tree referenced by the **Structure_I_DirectoryR** attribute of the structure object. Consider the following figure, which shows a directory record tree for a collection containing RRNs 18–30, where RRNs 20, 25, and 29 are not active.

Notes:

1. For the purpose of simplicity, our example assumes that a directory record only holds three directory entries. In reality, a directory record would hold many more directory entries.
2. Only directory entry fields for each directory record are shown. TPFCS makes use of other fields in the directory record to maintain information such as:
   - The level of the directory record (which is an integer 0 or greater, where a level of 0 indicates that the directory entries point directly to data or key records)
   - The number of directory entries currently in use
   - The RRN for which the directory record was created (which is usually the lowest RRN that this directory record locates).



* Contains the file address of both the primary as well as shadow data or key record for the RRN indicated.

Figure 38. How Directory Records Are Used to Locate RRNs 18–26

In Figure 38, the collection associated with directory records A–H is shadowed. Therefore, each of the directory records has both a primary and a shadow copy. For example, record A' is the shadow copy of record A. As with all of the records that

TPFCS uses, each shadow record contains the exact same contents as its primary counterpart. (To keep our discussion simple, the contents of the shadow records are not shown in our example and we will assume that TPFCS selects the primary copy.)

To understand how the tree of directory records is used to locate RRNs 18 and higher, consider a condition in which TPFCS needs to access RRN 24 of the associated collection. TPFCS arrives at the root of the tree via StructDasd_I_DirectoryR, which contains the file address of record A as well as the file address of its shadow, record A'. TPFCS reads record A **1** . Given the RRN we want to locate, TPFCS uses an internal algorithm to determine which RRNs each entry locates. This algorithm uses information such as the level of directory record A (which is 2), its number of entries (which is 3), and the RRN for which record A was allocated (RRN 18). In our example, the algorithm would determine that the first entry **2**, which points to record B, is used to locate RRNs 18–26 inclusive. TPFCS reads directory record B **3** and scans it using an algorithm to determine which entry locates RRN 24. In our example, the third entry **4** from the top of the record is used because it locates RRNs 24–26. TPFCS uses this directory entry to read record F **5** . Because directory record F is at level 0, each active entry points directly to a relative record in the associated collection. TPFCS uses the same algorithm used previously to determine that the very first entry **6** in the directory record contains the file addresses of the primary (and shadow) copy of the data or key record with RRN 24. Either file address is used to read information in the record with RRN 24, and both file addresses (when present) are used to update information in that record.

It is also important to note that not every RRN from 0 to the current maximum RRN used by a given collection will be in use. This is normally true when the operations performed by application programs on a given collection have caused a data or key record to be deleted as elements were first added to a collection and later on removed from that collection. For example, assume the current maximum RRN in use for the collection in question is 30. Notice how the directory entries corresponding to RRNs 20, 25, and 29 in records D, F, and G respectively do not contain file address information. Instead of containing file address information, they are marked as available. For the collection in question (whose current maximum RRN in use is 30), if TPFCS needs to allocate another RRN to store additional elements, it can use any of the RRNs that are marked available rather than allocating RRN 31.

You will also notice that in the directory records in Figure 38 on page 184, some of the entries are marked unused and others are marked as available. The difference between unused and available is as follows: Unused entries have never been used and contain zero. Available entries were previously used and once contained RRN file address information. The information formerly contained in these entries has been removed and the entries have been marked available as the associated RRNs were deleted. TPFCS can choose either unused entries or available entries to store file address information for new RRNs.

Whether or not a directory entry contains file address information is determined by its format. For a general discussion on directory entry format as well as how to determine more information about each format, see “Record Types” on page 181.

StructureDASD Attributes to Consider

The following are the main attributes from the StructureDasd object class that will be contained in the extended-residency structure object:

- **StructDasd_I_Directory**, which contains the file address of the structure record.

- Pointers to file chains for directory records, key records, and data records:
  - **StructDasd_I_AllocatedFACount_Data**, which is the count of records on the allocated data chain where TPFCS places data records.
  - **StructDasd_I_AllocatedFACount_Direct**, which is the count of records on the allocated directory chain where TPFCS places directory records.

    Note: Because it contains embedded directory entries, the structure record itself is linked as part of the allocated directory chain and is included in this count.
  - **StructDasd_I_AllocatedFACount_Index**, which is the count of records on the key (index) chain where TPFCS places key records.
  - **StructDasd_I_LastAllocatedFA_Data**, which contains the file address of the head of the allocated data chain.
  - **StructDasd_I_LastAllocatedFA_Direct**, which contains the file address of the head of the allocated directory chain.

    Note: Because the structure object contains embedded directory entries, the record containing the structure object itself is considered a special directory record. Consequently, when a collection is small enough to contain only embedded directory entries and no additional directory records, the structure object record itself is the head of the allocated directory chain and its file address will be contained in this field. When more elements are added to the collection to require additional directory records, the structure object record will be last in the chain.
  - **StructDasd_I_LastAllocatedFA_Index**, which contains the file address of the head of the allocated key (index) chain.

TPFCS places records on each of these chains in topdown order. This means that each chain is really a stack, and the head of each chain really points to the last record that TPFCS added on that chain rather than the first. That is why the name of each header contains the phrase *last allocated*. TPFCS always uses the TPF forward chain pointer of each record to point to the next element on the chain. The following figure shows how TPFCS adds a record to one of these chains:

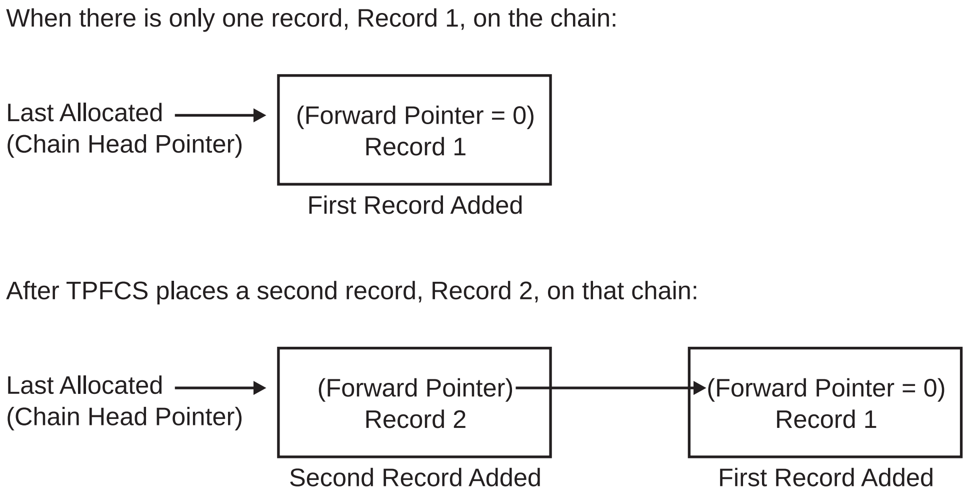


Figure 39. How TPF Collection Support Adds Records to a StructureDasd Chain

The allocated data and allocated key chains also contain records that TPFCS no longer uses to represent the collection associated with the StructureDasd object. As a result, when chasing these chains for the StructureDasd object of a given collection, not every record you find is necessarily being used by TPFCS for that collection. Rather than immediately removing records it no longer needs from the allocated data or allocated key chain, TPFCS chains these records to an associated *released chain* and schedules them to be released back to the TPF system when the length of the chain exceeds a predetermined threshold. The released chains are just another way of logically chaining the same physical records that are on the allocated chains.

TPFCS uses the following attributes of StructureDasd to manage each released chain:

- **StructDasd_I_ReleasedFACount_Data**, which is the count of records on the released data chain where TPFCS places data records that are scheduled for release.
- **StructDasd_I_ReleasedFACount_Index**, which is the count of records on the released key chain where TPFCS places key records that are scheduled for release.
- **StructDasd_I_LastReleasedFA_Data**, which contains the file address of the head of the released data chain.
- **StructDasd_I_LastReleasedFA_Index**, which contains the file address of the head of the released key chain.

Records are added to each released chain in topdown order similar to the way in which they are added to the allocated chains. TPFCS uses the TPF backward chain pointer to link records on each released chain. The following figure shows an example of how records are linked on both the allocated and released chains:

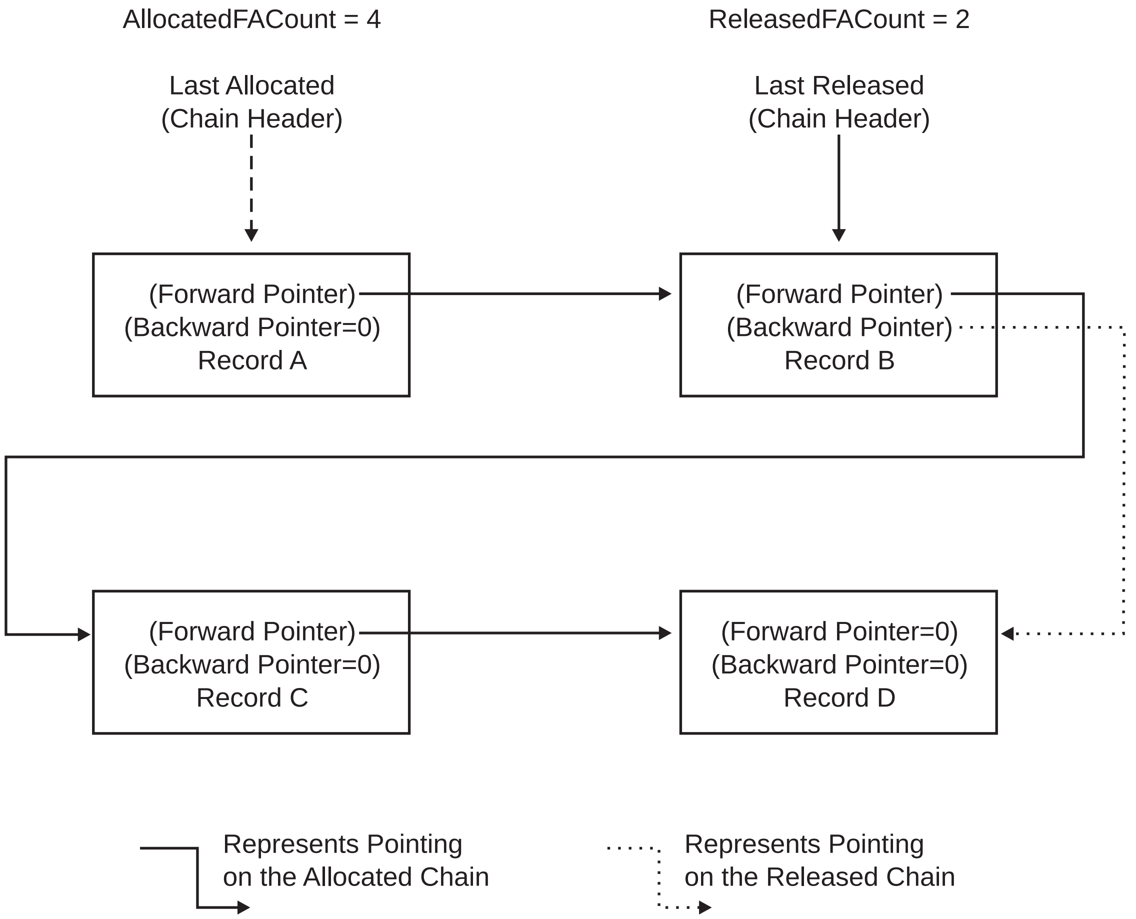


Figure 40. How Records Are Linked on Both Allocated and Released Chains

In Figure 40, the count of records on the allocated chain is 4 because the chain contains records A, B, C, and D. However, only records A and C are used for the collection because records B and D are on the released chain.

Note: TPFCS only uses the allocated chains and their counts for processing the validation and reconstruction requests available to you using the ZBROW command (see *TPF Operations* for more information).

To insert, retrieve, and delete elements from an extended-resident collection, TPFCS uses the following attributes:

- **StructDasd_I_Directory0** to **StructDasd_I_Directory17** contain the first 18 directory entries of the collection. These directory entries correspond to RRNs 0–17. Because these directory entries are located inside the extended structure (StructureDasd) object itself, we will refer to them as the *embedded directory entries*.
- **StructDasd_I_DirectoryR** is a directory entry that points to the root of a tree of directory records when all of the directory entries needed to represent the collection do not fit in the embedded directory entries.

DASDINDEX Structures

When TPFCS uses key records for a collection, its structure object will be a DASDINDEXPool object. DASDINDEXPool objects currently inherit attributes from the DASDINDEX class, which inherits from StructureDasd. The attributes are:

- **DASDINDEX_I_RootRRN** contains the RRN for the root of the key record tree structure.
- **DASDINDEX_I_RootDirEntry** contains a directory entry that points to the root key record as well as its shadow, if one exists.
- **DASDINDEX_I_MaxKeyLength** is the maximum key length allowed for a data element that TPFCS will store for this DASDINDEX object. The value of this attribute could be greater than the maximum length of a key for the actual collection as seen by an application programmer.

Figure 41 on page 190 is a conceptual diagram of a DASDINDEXPool structure object:

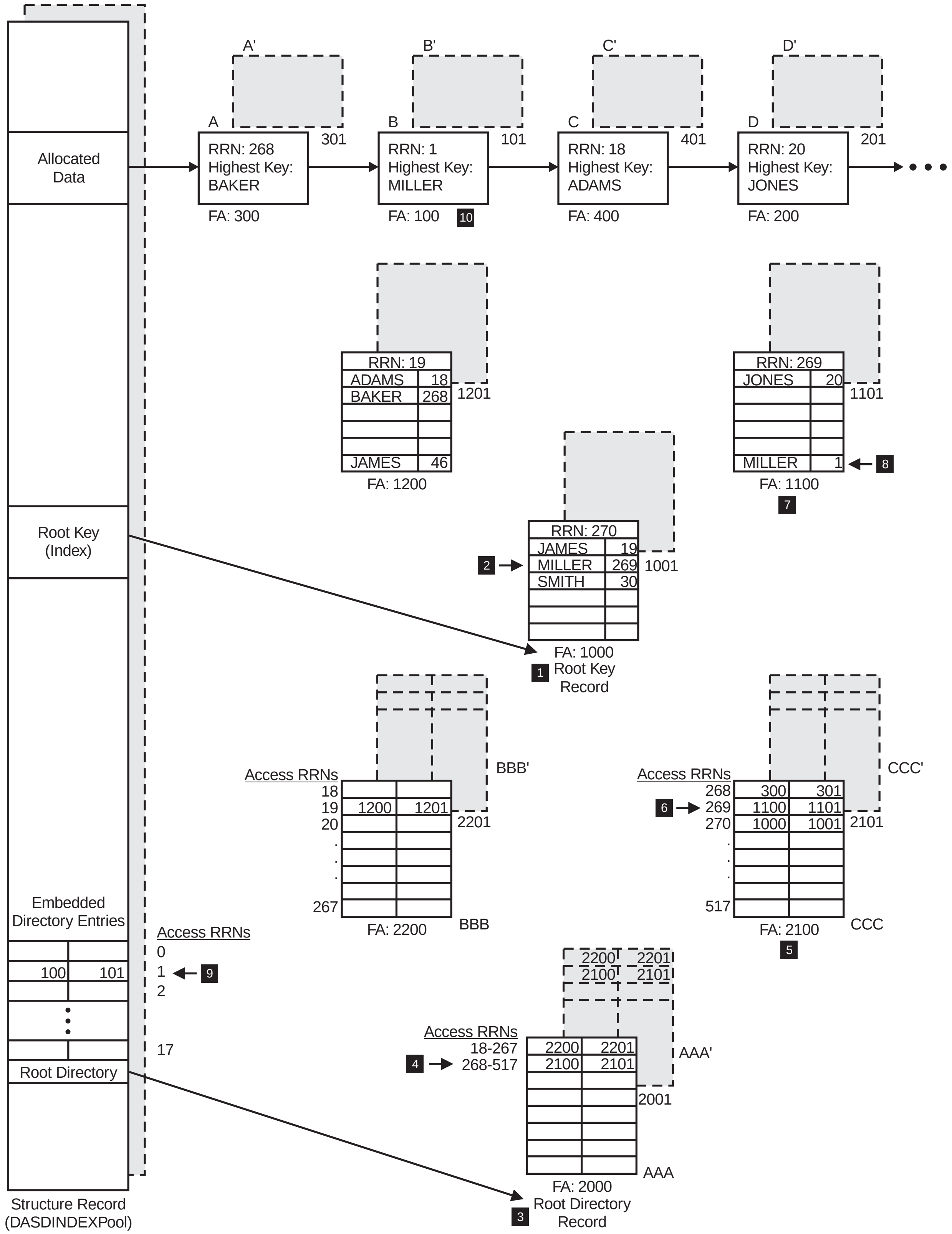


Figure 41. DASDINDEXPool Object and Its Associated Records

Note the following in Figure 41:

- The size and number of entries in each record, as well as file addresses shown, are not accurate. They are presented in such a way to make the figure as simple as possible.
- Similarly, the size of each record is not drawn to scale. All of the records are 4 K, including the record that contains the structure object (the DASDINDEXPool object).
- Not every field in the DASDINDEXPool object and not every field in the other records is shown; and not every key, data, or directory record is shown.
- A logical view and not a physical layout of all records is shown. For example, entries in key records are sorted using locators, but this is not shown.
- The allocated data field represents the StructDasd_I_LastAllocatedFA_Data attribute, which points to the allocated data chain.
- For the sake of simplicity, the released data chain is not shown. You can assume that none of the data records on the allocated chain happen to be released.
- The Highest Key label in each data record shows the highest value of a key for a data element stored in that data record. (Each data record can contain many data elements and their keys.) This label does ***not*** correspond to a special field in a data record.
- The RRN: label that is in each data and each key (index) record indicates the RRN of that record; that is, the relative record number for the directory entry that contains the primary and shadow file address for that record and its shadow. This label ***does*** refer to a special field in those records because TPFCS stores the RRN of every data or key record in the record itself.
- Every record in the figure has a shadow, which is represented with a dotted border. The contents of these records are not shown, but are always exactly the same as each primary record. Even the record containing the DASDINDEXPool object itself is shadowed.
- The Root Key (Index) label corresponds to DASDINDEX_I_RootDirEntry.
- The Embedded Directory Entries label corresponds to the directory entries which access RRNs 0–17. These directory entries are contained in the structure object as attributes from StructDasd_I_Directory0 to StructDasd_I_Directory17.
- The Root Directory label corresponds to StructDasd_I_DirectoryR, which contains the file address of the primary copy of the root of the directory record tree as well as its shadow copy.

To understand how the DASDINDEXPool object is used to represent a collection, consider an example in which TPFCS is processing a request of an application to retrieve the data element in the collection that has a key of LEMIE. TPFCS uses the root key pointer (DASDINDEX_I_RootDirEntry) to read the root key record (at file address 1000). **1** A key record contains one or more entries, each of which in turn contains a key value and an RRN. The entries are sorted by key. The root key record is scanned until either the first entry is found whose key is greater than or equal to the requested key (LEMIE), or the entries in the root key record are exhausted. In our example, TPFCS would stop scanning the root key record at the entry containing a key of MILLER and an RRN of 269 **2** .

TPFCS would then try to locate the directory entry for an RRN of 269. Because this RRN (269) is beyond RRN 17 (which is the highest RRN for an embedded directory), TPFCS begins its search for the directory entry to access that RRN by reading the root directory record of the directory record tree. The file address of the root Directory is contained in the root directory (StructDasd_I_DirectoryR) field in the DASDINDEXPool structure object. This address is 2000 in our example (record AAA) **3** . TPFCS reads the directory record (record AAA) from DASD and searches its entries using an internal algorithm to locate the entry that accesses RRN 269. The entry that TPFCS finds contains a primary file address of 2100 (as well as a shadow file address of 2101).[5] **4** TPFCS reads the record (record CCC) at this file address from DASD **5** .

Based on other fields in the record that are not shown, TPFCS knows that record CCC is another directory record, but that the file addresses contained in its directory entries point to key records, not directory records. TPFCS locates the directory entry in directory record CCC that accesses RRN 269 **6** . This entry has file addresses of 1100 and 1101, which point respectively to a primary and a shadow key record. TPFCS reads the key record at file address 1100 **7** (whose RRN is 269). TPFCS then scans this key record until it finds the first entry whose key is greater than or equal to the requested key of LEMIE, or until the entries are exhausted. In our example, the entry that satisfies this request contains MILLER as its key and an RRN of 1 **8** .

Once again, TPFCS attempts to locate the directory entry that accesses RRN. In this example, the search ends quickly because the directory entry that accesses RRN = 1 is the second embedded directory entry **9** . This directory entry points to the data record (record B) at file address 100 as well as its shadow (record B' at file address 101). TPFCS then reads this data record, whose highest key is MILLER **10** . TPFCS then searches this data record for a data element whose key matches LEMIE. If a match is found, the contents of the data element are returned to the requesting application. If no match is found, an appropriate error condition is returned.

We have just illustrated how, conceptually, the various components of a DASDINDEXPool structure object and the various records to which it points represent a keyed collection.

The following sections describe the contents of the structure record (where the DASDINDEXPool structure object is stored) as well as the other records used to store objects to manage the collection data. You will notice that none of these records contain DATXPAGE envelopes to store their objects.

Note: This is true even for the structure object, although it is a collection part. Recall that for extended-resident collections, the structure object is ***not*** embedded in the DATXPAGE envelope of the control record; rather, an xStructDASD object is embedded in place of the structure object itself. See “Collection Control Record” on page 170 for more information.

Structure Record

The structure record contains the following in the order listed:

1. The TPFCS record header as described under “TPFCS Record Header” on page 167. The record ID will be the ID of a directory record as defined in the ITO2 DSECT.
2. The structure object itself, which will be a DASDINDEXPool object.
3. There is no standard TPFCS record trailer (as described in “TPFCS Record Trailer” on page 169) at the end of a structure record containing a DASDINDEXPool object. However, the following information, normally contained in the trailer, will be included at the end of the record:
   - The file address of the owning control record

---

[5] This directory entry is not the final directory entry for RRN = 269 because the record to which it points on file is not a data or a key record. TPFCS will not stop searching for directory entries until the directory entry it locates points to a data or a key record.

- A reserved byte
- A TPF format flag.

Directory Records

The directory records contain the following in the order listed:

1. The TPFCS record header as described in “TPFCS Record Header” on page 167. The record ID will be the ID of a directory record as defined in the ITO2 DSECT.
2. The remainder of the record will be filled by a DIRECTORY object containing control information as well as actual directory entries. For the instance attributes of a DIRECTORY object, see the TPFCS source code.

   Note: The final bytes of a DIRECTORY object comprise a TPFCS record trailer as described in “TPFCS Record Trailer” on page 169. Because directory records themselves do not contain an RRN, TPFCS uses the field in the trailer for a directory record that normally contains an RRN to store an identifier. For an explanation of the contents of this identifier and how TPFCS uses it, see your IBM service representative.

Key Records

Key records will contain the following in the order listed:

1. The TPFCS record header as described in “TPFCS Record Header” on page 167. The record ID will be the ID of a key (index) record as defined in the ITO2 DSECT.
2. An NDXPAGE object, which is the object used to store keys and associated RRNs in the key record.

   Note: The NDXPAGE object fills the remaining portion of the record after the record header. Furthermore, it contains its own trailer, which will be located at the end of the record in place of the TPFCS record trailer.

The following example is a representation of a key record that contains five active entries. Note that the entries are sorted using locators in a similar way to how entries were sorted in a MemKey structure.

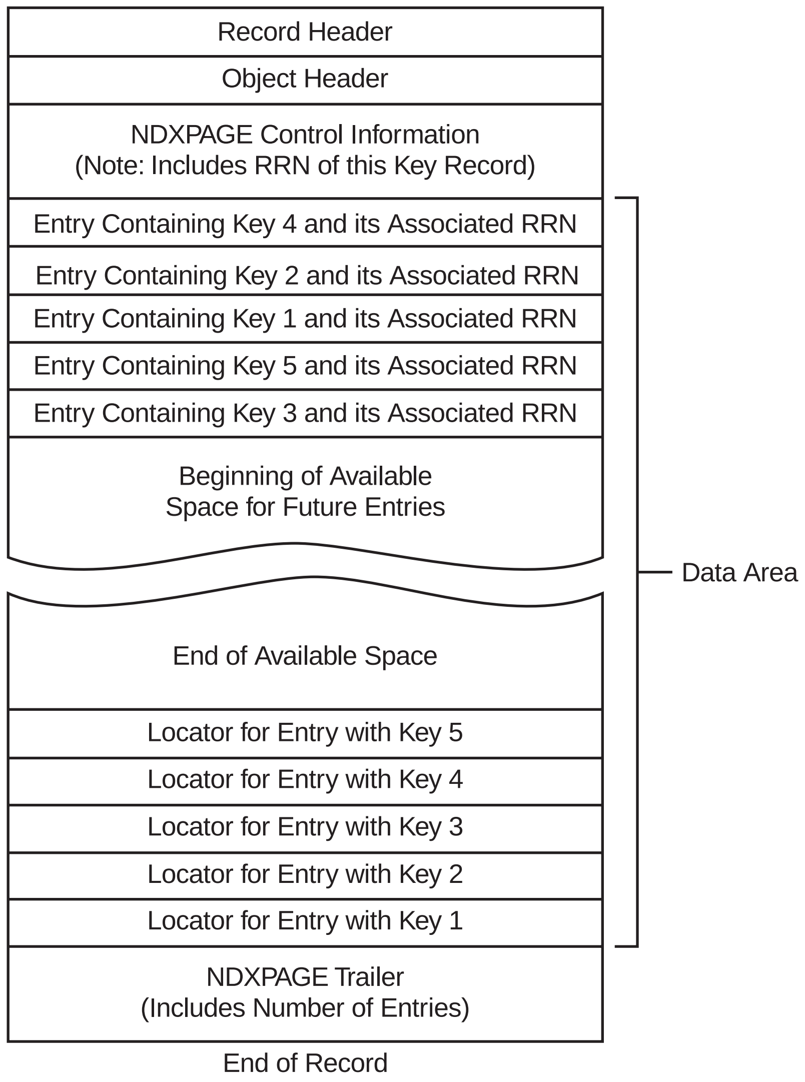

Record Header
Object Header
NDXPAGE Control Information
(Note: Includes RRN of this Key Record)
Entry Containing Key 4 and its Associated RRN
Entry Containing Key 2 and its Associated RRN
Entry Containing Key 1 and its Associated RRN
Entry Containing Key 5 and its Associated RRN
Entry Containing Key 3 and its Associated RRN
Beginning of Available
Space for Future Entries
Data Area
End of Available Space
Locator for Entry with Key 5
Locator for Entry with Key 4
Locator for Entry with Key 3
Locator for Entry with Key 2
Locator for Entry with Key 1
NDXPAGE Trailer
(Includes Number of Entries)
End of Record

Figure 42. Example of a Key (Index) Record Containing Five Entries

You can determine the exact format and contents of the NDXPAGE control information preceding its data area from the definition of the NDXPAGE instance attributes in the TPFCS source code. The exact format of the NDXPAGE trailer is defined under the NDXPG_Trailer tag in the same source module. The contents of this trailer include the following:

- The number of entries in use
- The displacement to the beginning of the available space in the data area relative to the start of the key record itself rather than the start of the NDXPAGE object
- The number of bytes in the available space.

Each entry in the data area consists solely of a unique key followed immediately by an associated RRN. There is no DSECT that defines these entries in the TPFCS source code because the length of an entry depends on the key length, which varies from collection to collection. The entry length is stored in the NDXPAGE locator for that entry and equals the collection key length plus the length of an RRN (4 bytes). If the NDXPAGE locators are corrupt, you can determine the collection key length in several ways:

- The collection key length is stored in DASDINDEX_I_MaxKeyLength, which will appear in response to a ZBROW COLLECTION command with the ATTRIBUTES parameter specified for the collection.
- The collection key length is also displayed in response to a ZBROW COLLECTION command with the DISPLAY parameter specified for the collection.
- If you suspect that the collection is corrupted and the response to these commands is incorrect, check the application program that defined and populated the collection.

The locators are used to sort the entries by key (from lowest to highest) as shown in Figure 42 on page 194. Each locator consists of 6 bytes in the following format:

- A 2-byte displacement field to the associated entry

  Note: For the key records, this displacement is relative to the beginning of the record (that is, starting at the TPFCS record header) rather than the NDXPAGE object or its data area.
- A 2-byte field containing the length of its associated entry
- 2 bytes reserved for future use.

Note: TPFCS does ***not*** remove the unused bytes from the data area before filing an NDXPAGE object in a key record on DASD.

Data Records

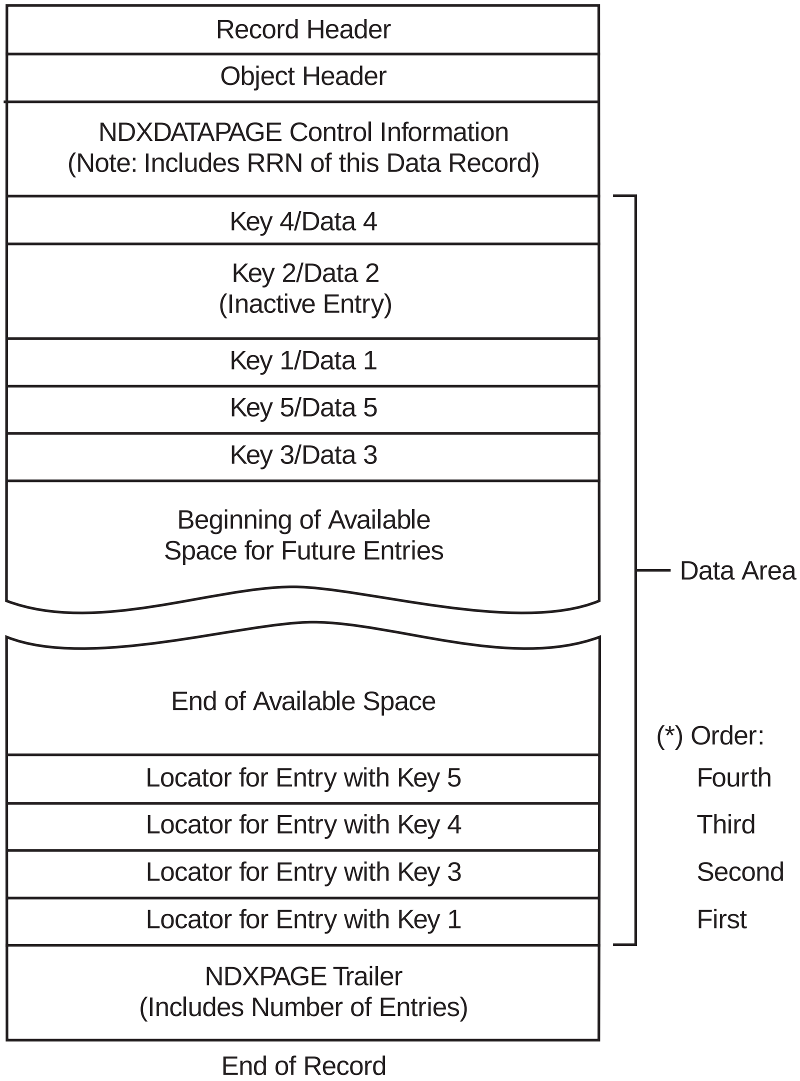
Data records contain the following in the order listed:

1. The TPFCS record header as described in “TPFCS Record Header” on page 167 . The record ID will be the ID of a data record as defined in the ITO2 DSECT.
2. An NDXDATAPAGE object, which is the object used to store data elements in the data records of collections with this structure.

   Notes:

   a. The NDXDATAPAGE object fills the remaining portion of the record after the record header. Just like the NDXPAGE object used for key records, the NDXDATAPAGE object contains an NDXPAGE trailer that will be located at the end of the record in place of the TPFCS record trailer.
   b. TPFCS does ***not*** remove the unused bytes from the data area before filing an NDXDATAPAGE object in a data record on DASD.

The following example is a representation of a data record that contains four active entries. As with key records, data record entries are sorted using locators.



(*) The order shown indicates the conceptual order as "seen" by applications that access the data: that is, the data with Key 1 is first, the data with Key 3 is second, and so on.

The entry that contains Key 2 has evidently been deleted and is, therefore, inactive because there is no locator that points to that entry.

Figure 43. Example of a Data Record Containing Four Active Entries

You can determine the exact format and contents of the NDXDATAPAGE control information preceding its data area from the definition of the NDXDATAPAGE instance attributes in the TPFCS source code. As mentioned previously, the trailer is the same as the NDXPAGE object and is described under the NDXPG_Trailer tag in the same source module.

Each entry in the data area consists of the following in the order indicated:

1. A 2-byte field containing the length of the USERdata object embedded in the entry where the data is located
2. The key for that entry

3. A USERdata object containing the data element in the associated collection that corresponds to that key.

Note: TPFCS only stores elements with unique keys in an NDXDATAPAGE object. As a result, if there are to be entries with duplicate keys in a data area of an NDXDATAPAGE object, it is most likely that all but one of those entries are remnants of data elements that have been deleted from the associated collection.

The format of the locators used to sort data record entries is the same as for those used to sort key record entries. See “Key Records” on page 193 for specific information.

DASDFLAT Structures

TPFCS uses DASDFLAT structure objects for the same types of collections that it uses for MemFLAT structures. Application programs specify a relative byte position when storing, updating, or retrieving data for those collections. As with MemFLAT structure collections, applications always specify a relative byte position when using DASDFLAT structure collections, either explicitly or as an index, depending on the type of collection the DASDFLAT structure represents.

To understand how DASDFLAT structure collections are stored in the TPF database, it is necessary to understand how TPFCS processes the relative byte position specified by the application to store or retrieve data for those collections. TPFCS takes the relative byte position, decrements it by 1 to make it 0-based, and then divides it by the number of bytes of data that would fit in a data record. TPFCS handles this quotient as an RRN and allocates a data record to hold the data when it is first stored in the collection. Directory entries are updated to locate the data record, and directory records are allocated as they are needed. For information about how directory entries and directory records relate to data records by RRN, see “Extended Structures (StructureDasd Class)” on page 181. If the data does not fit in one data record, an additional data record is allocated and the next sequential RRN is assigned to that record.

Note: If an application requests data from a relative byte position where data has never been stored, TPFCS will find an unused RRN for that position and will return a string of hexadecimal zeros to the requesting application for the number of bytes requested. TPFCS does not allocate any data records corresponding to gaps among the RRNs of the data records.

Structure Record

The structure record contains the following in the order listed:

1. The TPFCS record header as described on page 167. The record ID will be the ID of a directory record as defined in the ITO2 DSECT.
2. The structure object itself. The structure object will be a DASDFLATPool object, which inherits from the DASDFLAT class.
3. There is no standard TPFCS record trailer (as described in “TPFCS Record Trailer” on page 169) at the end of a structure record containing a DASDFLATPool object. However, the following information, normally contained in the trailer, will be included at the end of the record:
   - The file address of the owning control record
   - A reserved byte
   - A TPF format flag.

Directory Records

The directory records for a DASDFLATPool object are the same format as those used for DASDINDEXPool objects. See “Directory Records” on page 193 for more information.

Data Records

The data record for collections that use a DASDFLAT structure will contain a FLATPAGE object immediately following the TPFCS record header. The following are the important attributes of the FLATPAGE object:

- A data area where data for the collection is stored.
- Two 2-byte sequence counters that TPFCS uses to manage concurrent updates to logical records that span across several data records.
- The TPFCS record trailer.

Key Records

TPFCS does not use key records for collections of this structure type.

DASDHASH Structures

The best way to understand how a persistent collection is represented on the TPF database using a DASDHASH structure object is to consider the following algorithm:

TPFCS uses relative records 0–66 as anchor records for storing the collection data elements. This is done as follows:

- When an application requests that an element be stored in the collection, TPFCS takes the data element and performs a hash operation on it. It then takes the resulting hash value and divides it as an unsigned integer by 67.
- The remainder of this division (0–66) dictates the RRN of the data record where TPFCS will first attempt to store the data element. These data records will serve as anchor records.

TPFCS performs the following steps to store a data element in a data record:

Note: Initially, the current record is the appropriate anchor record.

1. The current data record is scanned to see if there is room for the element. If there is room in the current data record, the data element and its hash value are stored there.

   If there is no room in the current data record, TPFCS will check if there are any additional data records chained to it. If there is another in the chain that contains enough room, TPFCS will store the element and its hash value there.
2. If there are no additional records chained to the current data record or none of them have enough room, a new data record and RRN are allocated and the data element and hash value are stored in the new record. The new data record is chained to the current record using its RRN rather than its file address.

The following abstract model shows how this algorithm works:

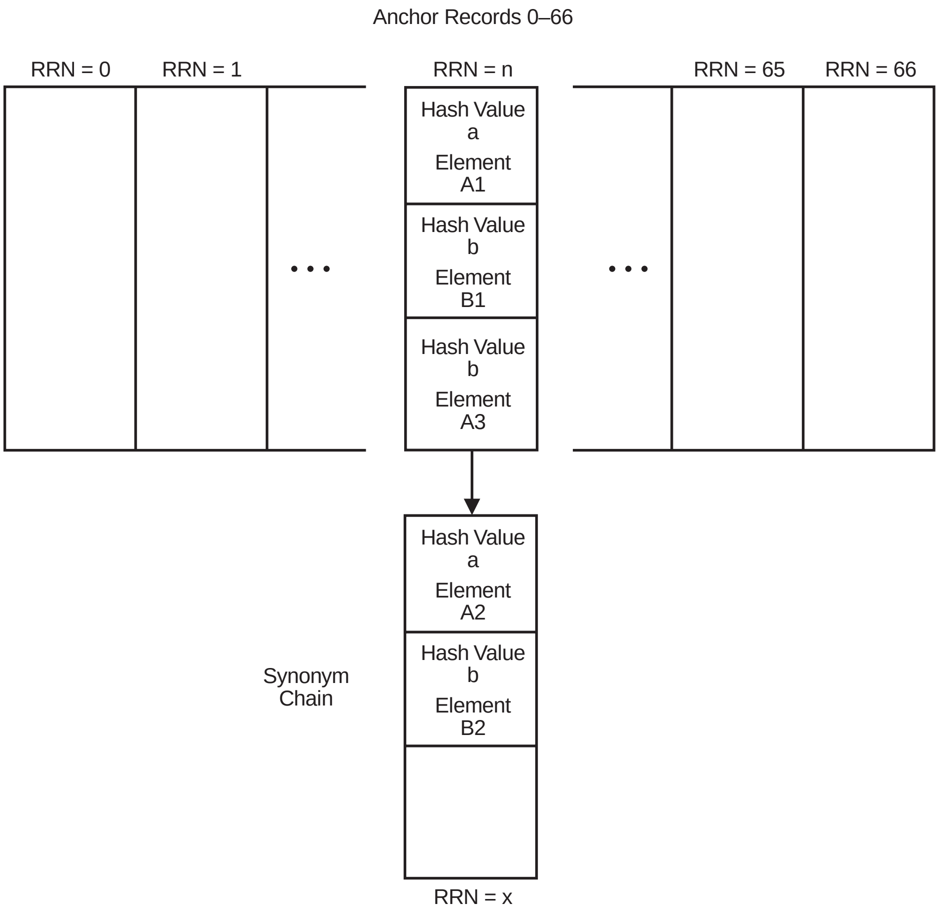


Figure 44. Abstract Model of a DASDHASD Structure Collection

In our model, elements A1, A2, and A3 all produce hash value a. Likewise, elements B1 and B2 produce Hash Value b.

- A1 was the first element that the application added to the collection. When Hash Value a is divided by 67, the remainder is n. Because there was room in the data record with RRN = n, element A1 and its hash value were stored there.
- B1 was the next element that the application added to the collection. When Hash Value b is divided by 67, the remainder is also n. Because there was room in the data record with RRN = n, element B1 and its hash value were stored there.
- Similarly, another element, C1, was also added to the collection. It had a unique hash value that, when divided by 67, produced a remainder of n. Element C1 and its hash value were, therefore, also stored in the data record with RRN = n.

  Element C1 does not appear in the figure because it was later deleted.
- When A2 and B2 were being added to the collection, another data record (with RRN = x) was used to store these elements and their hash values because there was no room for them in the data record with RRN = n.
- The application then removed element C1 from the collection.
- A3, with a hash value of a, was the next element to be added. Because the remainder of the division of its hash value is n, TPFCS first looked at the data record with this RRN to see if A3 and its hash value could be stored there. Because element C1 had been removed, there was enough room in that record, and TPFCS stored A3 and its hash value there.

Note: The data element and its hash value are not stored contiguously in the data record as shown in the figure. Figure 44 on page 199 is a conceptual representation of how the data elements and their corresponding hash values are stored. See the discussion of Figure 45 on page 201 for more information about how the data elements and their hash values are stored in a data record.

Structure Record

This record contains the following in the order listed:

1. The TPFCS record header as described on page 167. The record ID will be the ID of a directory record as defined in the ITO2 DSECT.
2. The structure object itself. The structure object will be a DASDHASHprime object, which inherits from the DASDHASH class.
3. There is no standard TPFCS record trailer (as described on page 169) at the end of a structure record containing a DASDHASHprime object. However, the following information, normally contained in the trailer, will be included at the end of the record:
   - The file address of the owning control record
   - A reserved byte
   - A TPF format flag.

Note: The DASDHASHprime and DASDHASH classes inherit the majority of their attributes from the StructureDASD class, and currently do not have any unique attributes of their own that are relevant to the level of our discussion. The most important attributes used to store data elements and retrieve them belong to the VDATPAGE object filed in the data record. See “Data Records” for more information.

Directory Records

The directory records for a DASDHASHprime object are the same format as those used for DASDINDEXPool objects. See “Directory Records” on page 193 for more information.

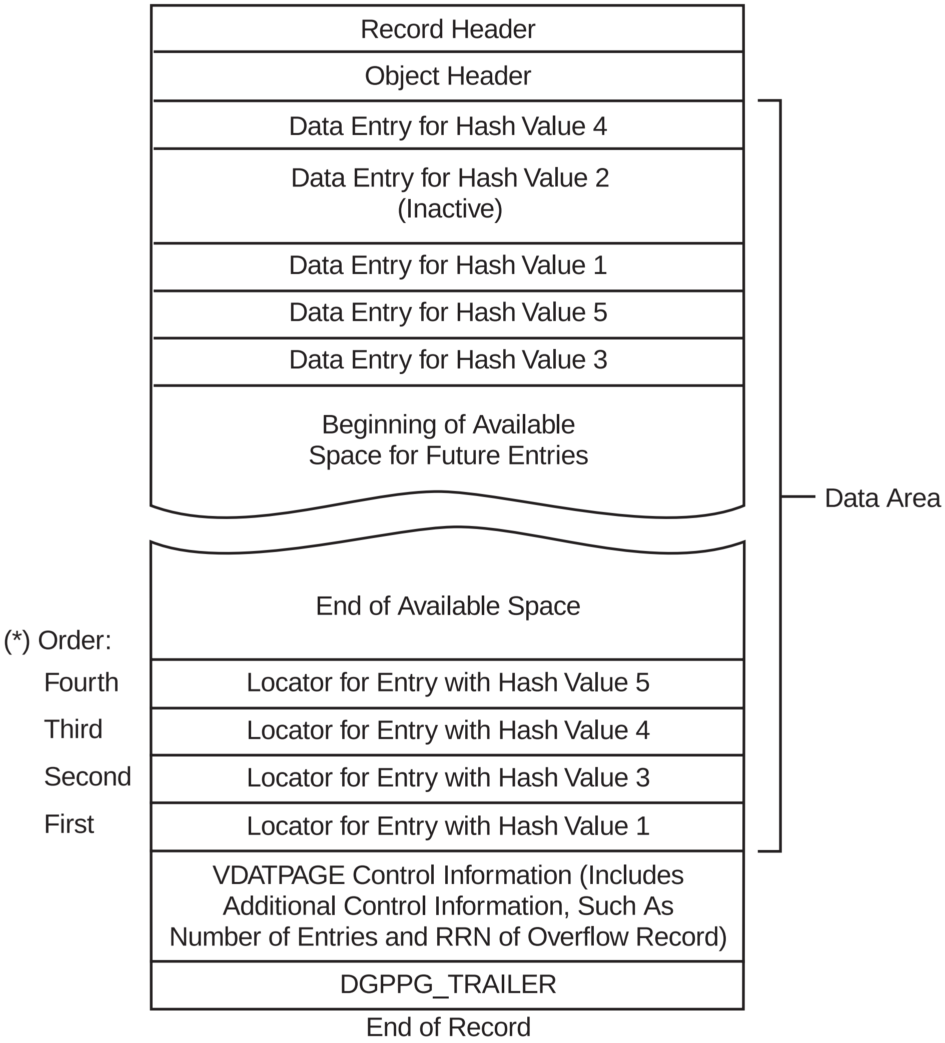
Data Records

These records contain the following in the order listed:

1. The TPFCS record header as described on page 167. The record ID will be the ID of a data record as defined in the ITO2 DSECT.
2. A VDATPAGE object, which is the object used to put data elements in a data record for collections with this structure.

   Note: The VDATPAGE object fills the remaining portion of the record after the record header. Furthermore, the VDATPAGE object includes the TPFCS record trailer.

The following figure represents a VDATPAGE data record that contains four active entries. Notice how the data entries are sorted by their associated hash values using locators that are at the bottom of the data area.


Record Header
Object Header
Data Entry for Hash Value 4
Data Entry for Hash Value 2 (Inactive)
Data Entry for Hash Value 1
Data Entry for Hash Value 5
Data Entry for Hash Value 3
Beginning of Available Space for Future Entries
Data Area
End of Available Space
(*) Order:
Fourth
Locator for Entry with Hash Value 5
Third
Locator for Entry with Hash Value 4
Second
Locator for Entry with Hash Value 3
First
Locator for Entry with Hash Value 1
VDATPAGE Control Information (Includes Additional Control Information, Such As Number of Entries and RRN of Overflow Record)
DGPPG_TRAILER
End of Record

(*) The order shown indicates the conceptual order of the data entries by hash value: that is, the Entry with Hash Value 1 is first, the Entry with Hash Value 3 is second, and so on.

The entry with Hash Value 2 has been deleted and is, therefore, inactive because there is no locator that points to that entry.

Figure 45. Example of a VDATPAGE Data Record Containing Four Active Entries

For more information about the control information attributes contained in the VDATPAGE object, see the definition for that object class in the TPFCS source code.

Each entry in the data area consists of the following in the order indicated:

1. A 2-byte field containing the length of the USERdata object embedded in the entry where the collection element is located.
2. A 2-byte field containing the length of the input value that was used to produce the data entry hash value. For some collections such as bags and sets, the input value is a replica of the data element itself.
3. The input value used to produce the hash value for this data entry.

4. A USERdata object containing the collection data element.

Each locator is an 8-byte field consisting of the following:

- A 4-byte field containing the hash value used to locate the associated data entry

  Note: The hash value is, therefore, ***not*** stored in the data entry itself.
- A 2-byte field containing the total length of its associated data entry
- A 2-byte displacement field to the associated entry.

Notes:

1. Unlike the locators used in the NDXDATAPAGE object for DASDINDEX structure collections described in "Data Records" on page 195, the displacement stored in each VDATPAGE locator is relative to the beginning of the VDATPAGE data area and ***not*** relative to the beginning of the data record.
2. TPFCS does ***not*** remove the unused bytes from the data area before filing a VDATPAGE object in a data record on DASD.

Key Records

TPFCS does not use key records for collections of this structure type.

DASDLIST Structures

TPFCS uses structure objects with DASDLIST attributes to represent collections where the elements have an order determined exclusively by the application. This differs from DASDINDEX structure collections where the order of collection elements is determined by the key associated with each element. For example, in a DASDINDEX structure collection, an element with a key of BAKER would naturally come before an element with a key of COOPER. On the other hand, in a DASDLIST structure collection, an element containing COOPER could precede an element containing BAKER because the application that created the collection has chosen to give COOPER a higher priority than BAKER.

To understand how TPFCS keeps track of arbitrarily ordered collections using the DASDLIST structure, consider the following scenario. Suppose that an application inserts the following elements into a collection in the order described, specifying the priority or order of each element as it is inserted:

1. The first element the application inserted contains BAKER.
2. The application then inserts COOPER in the collection, but specifies that COOPER is to be first in order (that is, to precede BAKER).
3. The application then inserts FOLEY in the collection, but specifies that FOLEY will now be first in order (that is, it will precede COOPER).
4. The application then inserts SMITH in the collection. However, this time the application specifies that SMITH should be last in the collection.

Let us assume that each of these elements the application inserted in the collection are large enough to each fill an entire data record. BAKER, the first element inserted, would be filed in relative record number (RRN) 0. COOPER, the next element inserted, would be filed in RRN 1, and so on. The following figure shows how each record that is written in the database would appear:

RRN 0	RRN 1	RRN 2	RRN 3
BAKER	COOPER	FOLEY	SMITH

Figure 46. How Elements of a DASDLIST Structure Collection Are Written in the Database

Notice that how the elements are ***written*** in the database differs greatly from a logical diagram of how those elements are ***ordered***, where FOLEY is the first element in the collection, COOPER is the second, and so on:

RRN 2	RRN 1	RRN 0	RRN 3
FOLEY	COOPER	BAKER	SMITH

Figure 47. How Elements of a DASDLIST Structure Collection Are Ordered

TPFCS uses fields in the DASDLIST structure as well as in the data records chained to that structure to keep track of the order of collections with arbitrarily ordered elements. The following are the fields in DASDLIST that are used to keep track of the order:

DASDLIST_I_FirstRRN
A 4-byte field that contains the RRN of the first data record in the collection, or −1 if the collection is empty.

DASDLIST_I_LastRRN
A 4-byte field that contains the RRN of the last data record in the collection, or −1 if the collection is empty.

Note: For other attributes of the DASDLIST structure object, such as the number of elements in the associated collection, see the definition of the DASDLIST instance attributes in the TPFCS source code.

TPFCS chains the data records in order as well as in reverse by maintaining the following fields in each of the data records:

LDATPAGE_NEXTRRN
A 4-byte field that contains the RRN of the next data record as ordered by the application, or −1 if this is the last data record for the collection.

LDATPAGE_PREVRRN
A 4-byte field that contains the RRN of the previous data record as ordered by the application, or −1 if this is the first data record for the collection.

Figure 48 shows how the DASDLIST structure object, together with the next RRN and previous RRN fields in each of the data records, enables TPFCS to traverse the collection either in order from the first to the last element, or in reverse order from the last element to the first element:

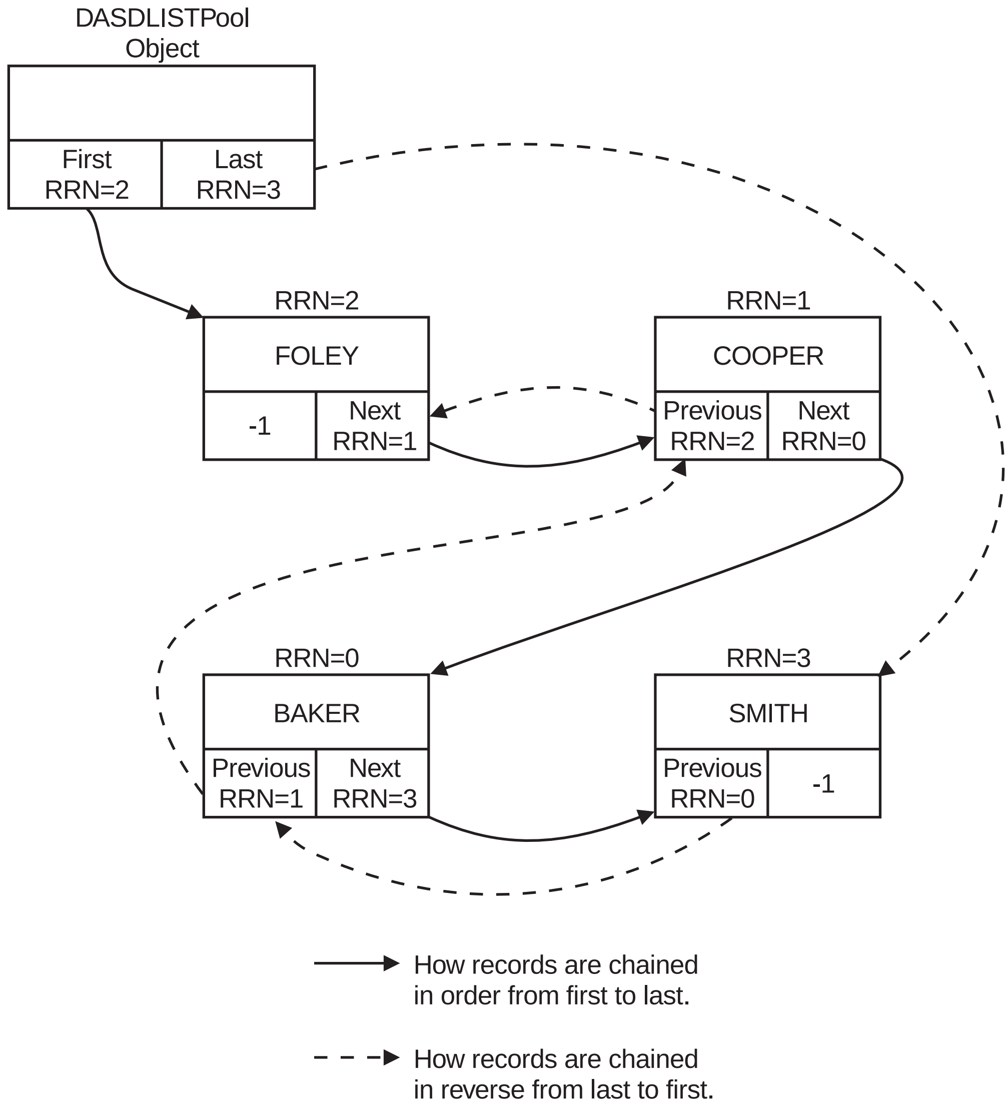


Figure 48. How the Order of a DASDLIST Collection Is Maintained

By using the DASDLIST_I_FirstRRN structure object field (labeled First RRN in the figure), TPFCS can locate the first element in the collection regardless of where it is filed in the TPF database. (For more information about how directory entries and directory records are used to locate the data record corresponding to a given RRN, see Figure 38 on page 184 and the discussion that follows it). TPFCS can then traverse the entire collection, in order, using the LDATPAGE_NEXTRRN field (labeled Next RRN in the figure) in each data record. Similarly, TPFCS can traverse the entire collection in reverse order, starting with the last element, by using the DASDLIST_I_LastRRN structure object field (labeled Last RRN) and the LDATPAGE_PREVRRN field (labeled Previous RRN) in each data record.

In reality, not every element in a DASDLIST structure collection will fill an entire data record. In a real example of a collection, each of the data records of a DASDLIST structure collection would contain many elements. TPFCS manages the order of elements in these data records by using locators to sort them based on how they were ordered by the applications that added them to the collection. The contents of a data record for DASDLIST structure collections are, therefore, more complex than the example we have just discussed, and data records will contain many more fields to count and sort the elements they contain. See “Data Records” for a complete discussion of the format of data records for DASDLIST structure collections.

Structure Record

This record contains the following in the order listed:

1. The TPFCS record header as described on page 167. The record ID will be the ID of a directory record as defined in the ITO2 DSECT.
2. The structure object itself. The structure object will be a DASDLISTPool object, which inherits from the DASDLIST class.
3. There is no standard TPFCS record trailer (as described in “TPFCS Record Trailer” on page 169) at the end of a structure record containing a DASDLISTPool object. However, the following information, normally contained in the trailer, will be included at the end of the record:
   - The file address of the owning control record
   - A reserved byte
   - A TPF format flag.

Directory Records

The directory records for a DASDLISTPool object are the same format as those used for DASDINDEXPool objects. See “Directory Records” on page 193 for more information.

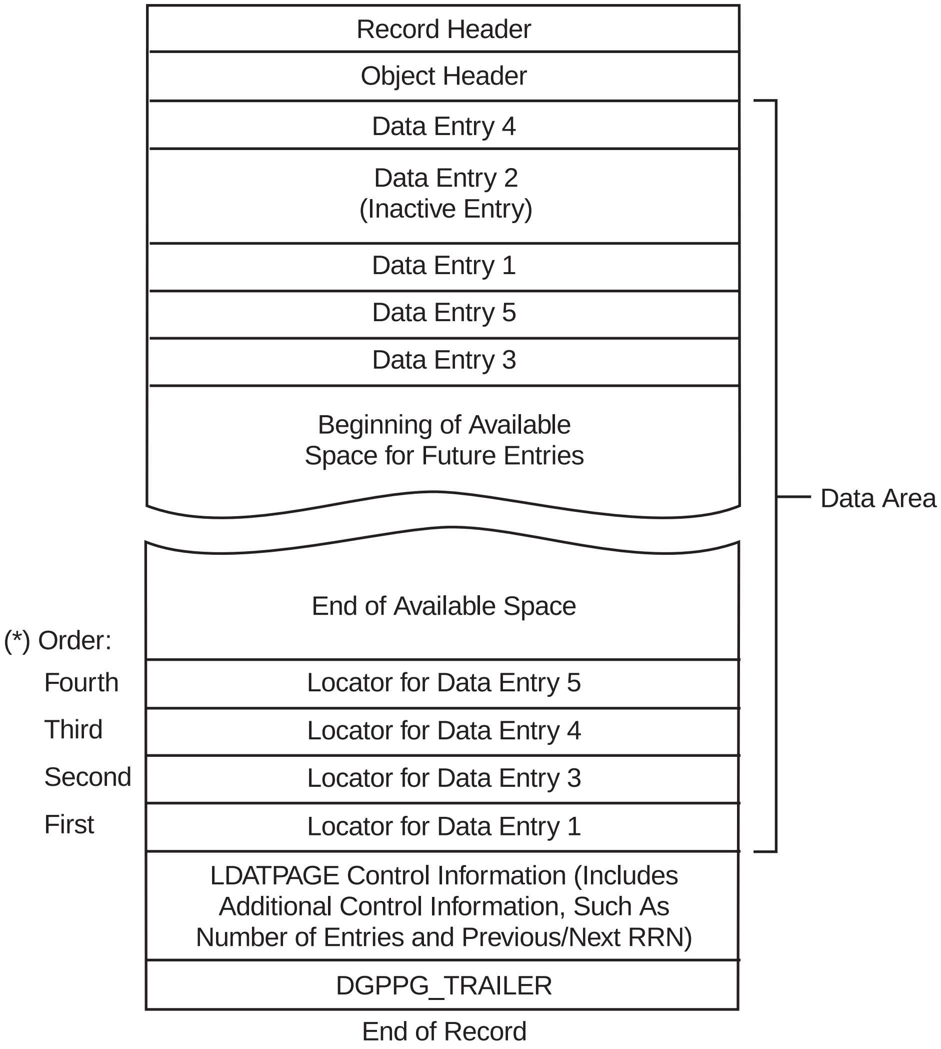
Data Records

These records contain the following in the order listed:

- The TPFCS record header as described in “TPFCS Record Header” on page 167. The record ID will be the ID of a data record as defined in the ITO2 DSECT.
- An LDATPAGE object, which is the object used to fill data elements in a data record for collections with this structure.

  Note: The LDATPAGE object fills the remaining portion of the record after the record header. Furthermore, this object includes the TPFCS record trailer.

The following example is a representation of a data record that contains four active entries. Notice how the data entries are sorted using locators, which are at the bottom of the data area.



(*) The order shown indicates the conceptual order as "seen" by applications that access the data: that is, Data Entry 1 is first, Data Entry 3 is second, and so on.

Data Entry 2 has evidently been deleted and is, therefore, inactive because there is no locator that points to that entry.

Figure 49. Example of an LDATPAGE Data Record Containing Four Active Entries

We have already discussed two items contained in the LDATPAGE control information: namely, the RRN of the previous data record and the RRN of the next data record in the collection. You can determine the exact format and contents of all the LDATPAGE control information preceding its data area from the definition of the LDATPAGE instance attributes in the TPFCS source code.

Each entry in the data area consists of the following in the order indicated:

1. A 2-byte field containing the length of the USERdata object embedded in the entry where the data is located.
2. A USERdata object containing the data element stored in that entry.

Each of the locators used to maintain order among the data entries is a 4-byte field containing the following:

- A 2-byte field containing the length of its associated entry
- A 2-byte displacement field to the associated entry.

Note: Unlike the locators used in the NDXDATAPAGE object for DASDINDEX structure collections described in “Data Records” on page 195, the displacement stored in each LDATPAGE locator is relative to the beginning of the LDATPAGE data area and ***not*** relative to the beginning of the data record.

Key Records

TPFCS does not use key records for collections of this structure type.

How Objects Are Stored on DASD

All objects that TPFCS files to DASD are written in 4-K pool records. As mentioned previously, the objects used by TPFCS are filed to DASD in a slightly different format from their representation in memory when they are actually used. The general nature of these differences is as follows:

- **Space Saving**: Many objects TPFCS uses to hold collection data elements contain attributes that are contiguous data areas. For a given collection, TPFCS stores collection data elements in the object data area.

  The object data area is allocated when the object is first created and is filled as applications add elements to the associated collection. The number of bytes of the object data area currently in use is, therefore, dynamic. Sometimes, for compact structure objects only to save DASD space, TPFCS will strip the object of those bytes contained in the object data area that are not in use before filing the object in the TPF database. See “How Some Objects Are Condensed to Save Space” for more information.
- **Packaging:** Before being written to DASD, objects which TPFCS includes in a collection control record are inserted into a DATXPAGE object. The DATXPAGE object functions as an envelope to hold those objects. Packaging was discussed in detail in “Packaging in DATXPAGE Envelopes” on page 169.
- **Overflow:** TPFCS files all of its objects in 4-K records. Because the objects belonging in a collection control record may not fit in a single 4-K record, they may overflow into one or more 4-K additional records. See “How an Object Can Overflow into Additional Records” on page 210 for more information about how TPFCS handles overflow conditions.

How Some Objects Are Condensed to Save Space

As mentioned previously, TPFCS sometimes condenses compact structure objects that contain data areas by removing unused portions of the data area before filing those objects. TPFCS condenses such objects to save DASD space. When the object is filed on DASD, it appears as follows:

- The displacements to any fields that were relocated will differ from their definition in the TPFCS source code and will vary.
- The length field in the object header, as well as any attributes that represent the length of the data area, reflect the number of bytes that will be contained in the object when TPFCS reads it into memory, and will be greater than the number of bytes contained in the object as filed on DASD.

Figure 50 clarifies these points. Consider the following object as it appears in memory as the TPF system accesses it to process application requests against an associated collection:

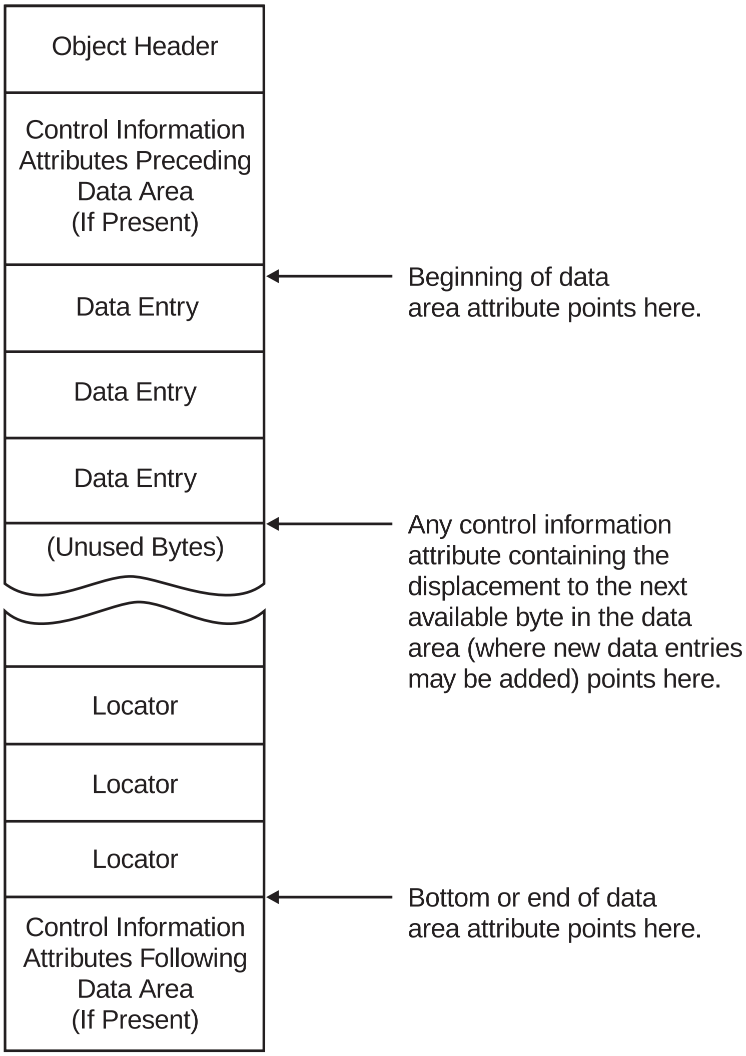

Object Header
Control Information Attributes Preceding Data Area (If Present)
Data Entry
Data Entry
Data Entry
(Unused Bytes)
Locator
Locator
Locator
Control Information Attributes Following Data Area (If Present)
Beginning of data area attribute points here.
Any control information attribute containing the displacement to the next available byte in the data area (where new data entries may be added) points here.
Bottom or end of data area attribute points here.

Figure 50. How an Object Containing a Data Area Appears in Memory

Each data entry is a field in the data area where collection elements are stored. In objects such as these, the collection elements need to be sorted. As was discussed previously for compact structures, TPFCS normally inserts elements that need to be sorted into the data area as they arrive, and locators are used to retrieve the elements in correct order as if they were sorted.

Also, as discussed previously, when using locators, TPFCS inserts data entries starting at the top of a data area working downward, and adds locators to the same data area starting at the bottom and working backward. This practice leaves a gap of unused bytes between the last data entry and the displacement where the locators are stored. This gap of unused space is where additional entries are added as applications add or insert more elements in the associated collection.

Recall that objects with locators contain control information attributes as well as a data area. Depending on the specific object, these attributes can occur before the data area, after it, or at both locations. The control information attributes determine information such as the entire size of the data area (including unused bytes), the number of unused bytes, the displacement in the data area where the unused bytes begin, and so on. TPFCS will use this information to determine whether there is room for the data entry, as well as where in the data area to store the entry, when inserting an additional data entry into the object data area. Besides having the length of its data area in one of its control information attributes, the object has its entire length stored in the object header.

Figure 51 shows how the same object is condensed for space saving when it is filed in a record on DASD:

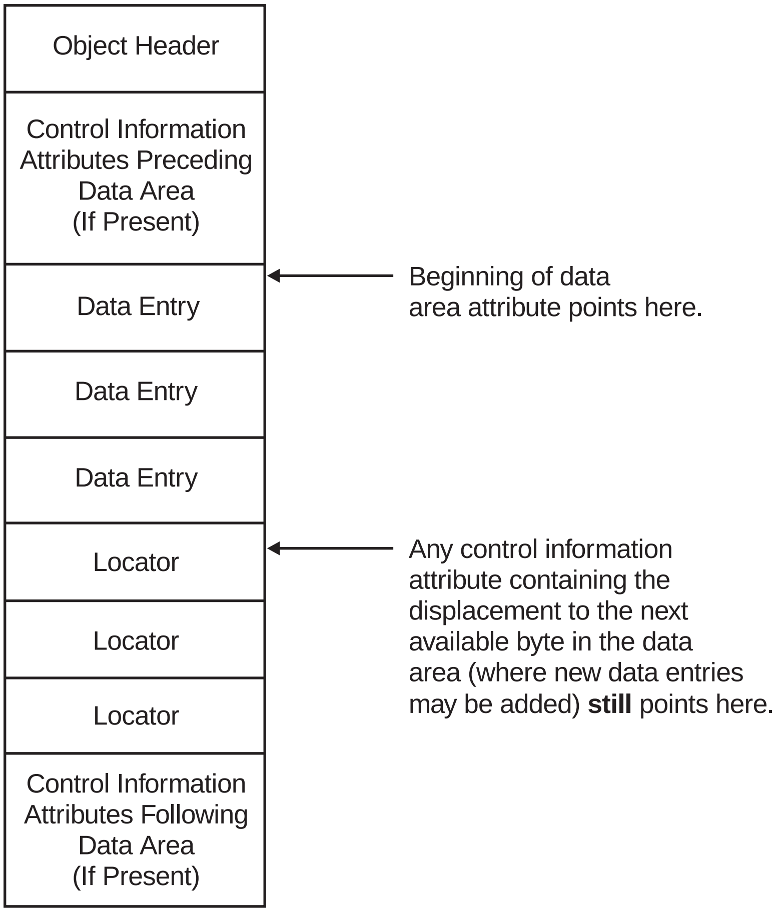

Object Header
Control Information Attributes Preceding Data Area (If Present)
Data Entry
Beginning of data area attribute points here.
Data Entry
Data Entry
Locator
Any control information attribute containing the displacement to the next available byte in the data area (where new data entries may be added) **still** points here.
Locator
Locator
Control Information Attributes Following Data Area (If Present)

Figure 51. How an Object Containing a Data Area with Locators Appears after Space Saving

The data area of the object is now much smaller because the gap of unused bytes between the last data entry and the beginning of the locators is no longer present. This is the ***only*** difference between how the object appears in memory and how it is stored on DASD. For example, the contents of the following are ***not*** changed:

- The object header contains the length of the entire object when it is active in memory even though the object, as it appears on DASD, is smaller because TPFCS removed the unused bytes from its data area. Likewise, all control information attributes used to contain length values, such as the length of the data area, have not changed and reflect the length of the data area in memory (which is larger) rather than the length of the data area on DASD. TPFCS maintains the original values for all length attributes because this information is needed to expand the object when reading it into memory.

- All other fields in the control information attributes or the locator fields are ***not*** changed. These fields can include displacements that are relative to the object as it appears in memory, ***not*** as it appears on DASD. An example of such a field is an attribute containing the displacement in the data area where the unused bytes begin. TPFCS does ***not*** reset these fields because they are needed when the object is read and expanded in memory to its original size.

Note: The only objects that TPFCS condenses on DASD for space saving are those indicated in “Compact Structures (StructureMem Class)” on page 173. Objects filed in data records listed under “Extended Structures (StructureDasd Class)” on page 181 are ***not*** condensed.

How an Object Can Overflow into Additional Records

As mentioned previously, not every collection part that TPFCS wants to file for a given collection can fit in a single DATXPAGE envelope in the control record because this envelope itself must fit in a 4-K record. When this occurs, the object splits and overflows into one or more additional DATXPAGE envelopes, each of which is filed in a separate 4-K record.

When an overflow condition occurs, TPFCS files another object (called an *xternalObject*) in the very first 4-K record along with the beginning of the object. The xternalObject serves as a pointer that contains the file addresses of all the 4-K records that TPFCS needed to use to file a single object or set of related objects. Once again, each *portion* of the object that is filed in each record is packaged in a separate DATXPAGE envelope.

Note: Do not confuse a ***portion*** of an object with an ObjectPart (collection part). A collection part or ObjectPart is itself an ***entire*** object, which can be divided into several portions and filed in separate records. OBJECTA in Figure 52 on page 211 is most likely a collection part object.

The following figure shows how an object is split into portions and packaged into several DATXPAGE envelopes to be filed in more than one 4-K record:

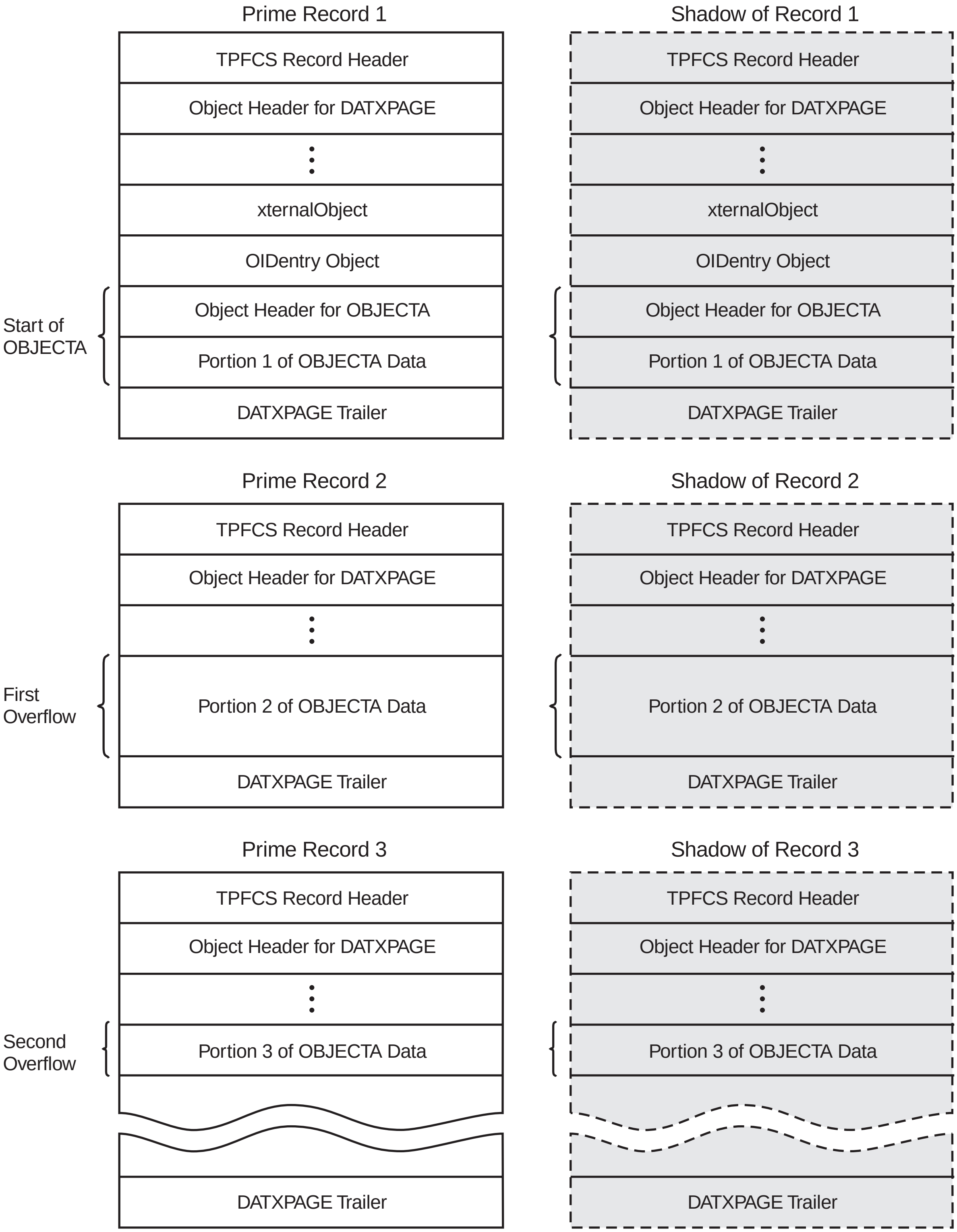


Figure 52. How OBJECTA Is Spread Across Three Records (with Shadowing)

Note: When an object such as OBJECTA does not fit in a single record, it can split at any location. It could also have split in the middle of its object header. Furthermore, the split could have occurred in the middle of the OIDentry object rather than in OBJECTA. (These scenarios are not shown for the sake of simplicity.)

The previous example could also have contained more than a single object in place of OBJECTA. For example, we could have been dealing with objects called OBJECT1, OBJECT2, and OBJECT3. In this example, TPFCS would have attempted to file all three objects in sequence, one immediately after the other starting with OBJECT1, in a single DATXPAGE envelope and corresponding 4-K record. As soon as the first DATXPAGE envelope and its corresponding record was filled, TPFCS would continue with the current object and begin to fill another envelope in another record until all of the objects have been accounted for. By the time TPFCS filed all of the objects, it could have filed them in separate records, but this does not imply that each object would be contained entirely in its own record.

Not every object will fit in a single 4-K record along with all of the other objects TPFCS chooses to file there for the same collection. For example, for a compact-resident collection, the structure object, which houses the data elements of that collection, can overflow into additional records that are beyond the control record. The following figure shows such an example, where the structure object for a compact-resident array collection overflows into another 4-K record. When this occurs, an xternalObject is included in the first DATXPAGE envelope and points to the next and any subsequent records into which an overflow has occurred.

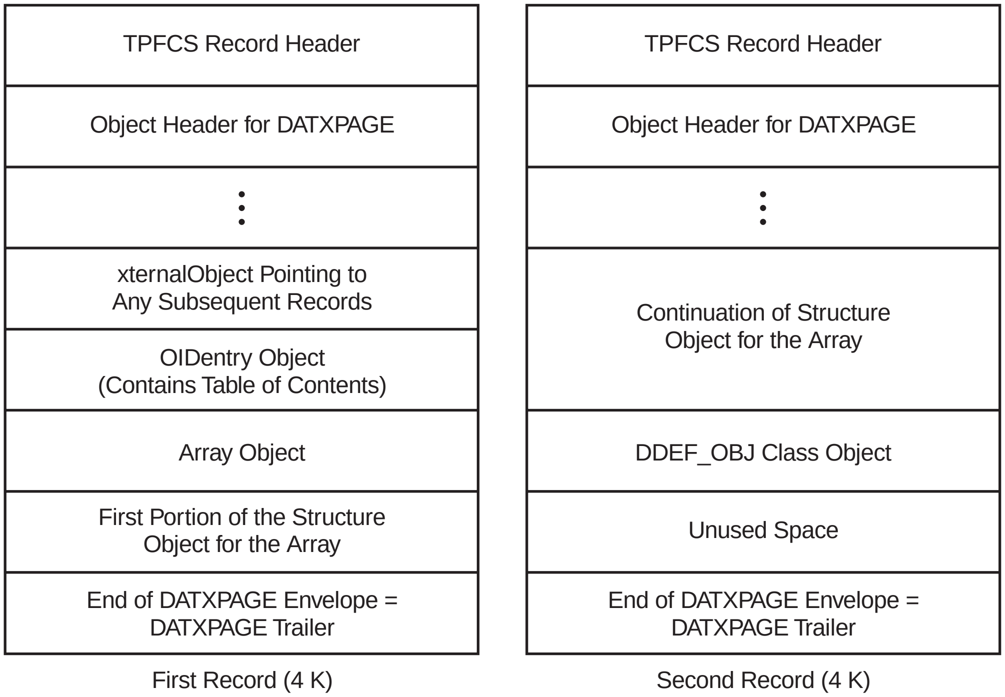

TPFCS Record Header
Object Header for DATXPAGE
xternalObject Pointing to Any Subsequent Records
OIDentry Object (Contains Table of Contents)
Array Object
First Portion of the Structure Object for the Array
End of DATXPAGE Envelope = DATXPAGE Trailer
First Record (4 K)
TPFCS Record Header
Object Header for DATXPAGE
Continuation of Structure Object for the Array
DDEF_OBJ Class Object
Unused Space
End of DATXPAGE Envelope = DATXPAGE Trailer
Second Record (4 K)

Figure 53. Control Record for a Nonshadowed Array with Overflow

It is important to realize that TPFCS will handle both physical records as a single logical control record when reading them into memory to service an application request to access the collection. Because of this, all displacements in the OIDentry table of contents are relative to the beginning of the OIDentry when all of the objects of our previous example are read into one contiguous buffer with the surrounding DATXPAGE envelopes removed as follows:

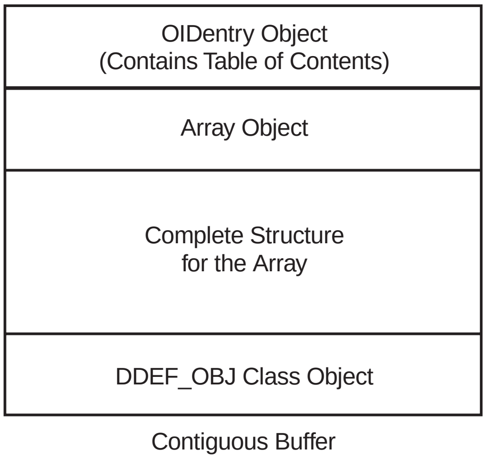


Figure 54. How TPFCS Handles the Control Record Contents

In addition, any other object attributes or fields containing pointers for the logical control record refer only to the *first* physical 4-K record in which control record information is written: that is, the record containing the xternalObject. For example, the PID of the array whose control record is shown in Figure 53 on page 212 would contain the file address of First Record. This same file address is stored in the *owner ID* discussed below.

xternalObject

As discussed earlier, an xternalObject is placed in a file record when an object does not fit in a single control record and has overflowed into one or more additional records.

Note: In our discussion, we are still referring to the example described in Figure 52 on page 211.

The following are attributes or fields to look for in a filed xternalObject:

- XOBJ_OID, which is a field containing the persistent identifier (PID) of the collection for which TPFCS uses that object. (In our example, XOBJ_OID would contain the PID of the collection associated with OBJECTA.)
- XOBJcount, which is the count of overflow records, including the first. (In our example, XOBJcount would contain three records.)
- One XOBJentry for each primary record. (In our example, there would be three XOBJentry fields, which include an entry that points to both the primary and shadow copies of record 1.) Each XOBJentry contains:
  - A directory entry containing the file addresses of both the primary as well as the shadow copy of the record if one exists. The format of directory entries are also used elsewhere by TPFCS. Unless otherwise noted, their format corresponds to the first directory entry format discussed in “Record Types” on page 181.
  - The record ID and record code check (RCC) of the record.

You can calculate the exact displacements to these attributes as well as their lengths by reading the definition of the xternalObject instance in the TPFCS source code.

Owner ID

The first pool record containing the primary control record is considered the *owner* of all other collection parts that are chained together to form the internal collection structure in the TPF database. This pool record is often referred to as the *owning control record*, and its file address, as well as additional control information, is stored in a field as either the *owner identifier* (*ID*) or *owning ID* in the TPFCS source code. This ID is often stored in many of the other pool records that are written to the TPF database as part of the TPFCS internal representation of a given collection.

Determining Where Collections Are Stored on DASD

Using commands, you can display information about collections and determine where different components of a collection reside on DASD.

Locating the Collection Control Record

The control record (like all of the other records TPFCS uses to represent a given collection) is a pool record. As mentioned previously, the PID contains the file address of the prime control record and, if the collection is shadowed, it contains the file address of the shadow control record too. See the definitions for the OIDnumber class hierarchy in the TPF source code to determine the location of the file addresses in the PID.

Listing Collection Parts

Enter the ZBROW COLLECTION command with the PARTS parameter specified to display a summary of the part objects that comprise a specified collection and which are located in its logical control record. For example, you can analyze an array collection as follows:

```
ZBROW COLLECTION PARTS ARRAYCOL
BROW0602I 14.21.50 BROWSER QUALIFIED FOR DSNAME TO2SVTDS
BROW0403I 14.21.50 COLLECTION PARTS DISPLAY
 PART    NAME
    0    OIDentry
    1    ARRAY
    2    DASDFLATPool
    3    DDEF_OBJ_DASD
END OF DISPLAY
BROW0410I 14.21.50 BROWSE OF COLLECTION COMPLETED
```

Similarly, a collection that consists of multiple structures such as represented in Figure 31 on page 172 might be represented as follows:

```
ZBROW COLLECTION PARTS KEYEDLOGCOL
BROW0602I 14.41.26 BROWSER QUALIFIED FOR DSNAME TO2SVTDS
BROW0403I 14.41.26 COLLECTION PARTS DISPLAY
 PART    NAME
    0    OIDentry
    1    KeyedLog
    2    DASDFLATPool
    3    DDEF_OBJ_DASD
    4    KeySet
    5    MemKey
    6    DDEF_OBJ_DASD
END OF DISPLAY
BROW0410I 14.41.26 BROWSE OF COLLECTION COMPLETED
```

Displaying Collection Part Contents

You can display the contents of these object parts, otherwise known as their attribute values, by using the ZBROW COLLECTION command with the ATTRIBUTES parameter specified for a given collection. This display lists the object parts that are used to represent the specified collection and many of the attributes of the parts, including attributes that those objects inherit from higher classes. [6] The data and control information for the collection are stored differently for each class of object and can require several screens to be fully displayed.

The collection attributes display, which results from the ZBROW COLLECTION command with the ATTRIBUTES parameter specified, has the following format. The header of this display includes the collection type (collection class name) for the specified collection. Each collection part follows, with the name of the part displayed. This is the same list of parts that would result from the ZBROW COLLECTION command with the PARTS parameter specified. Remember that each collection part is an object. In each part display, the class hierarchy for the object's class is shown, with the name of each class displayed, starting with the root class (OBJECT). These are the same classes that would be displayed using the ZBROW CLASS command with the TREE parameter specified. In each class display, there is a list of the attributes for that class and a list of the attribute values for that object.

Figure 55 shows the format for the collection attributes display. The figure shows the layout for one part. An actual display could contain any number of collection part objects, each of which inheriting from any number of higher object classes.

---

[6] Not every attribute of an object is displayed by this command. Which attributes are displayed, as well as the order in which they are displayed, does not correspond to the order in which they will be seen when displaying the record where that object has been filed. The attributes displayed and their order is determined by the presence and order of the PRINTATTRIBUTE macros for the definition of the class of the object in the TPFCS source code. Look at the order in which the instance attributes are defined in the TPFCS source code to determine their order in a file record.

```
ZBROW COLLECTION ATTRIBUTE collection_name 1
BROW0406I 15.03.08 COLLECTION ATTRIBUTES DISPLAY
ATTRIBUTE                           VALUE
COLLECTION CLASS NAME - collection_type 2
  PART  NAME  **********            PartA 3
CLASS NAME  **                      OBJECT 4
OBJECT_ID                           PartA ID
OBJECT_LGH                          PartA length
OBJECT_SEQ_CTR                      PartA sequence counter
CLASS NAME  **                      ObjectPart
⋮
  (collection_name attributes inherited from class ObjectPart)
⋮
CLASS NAME  **                      ObjectA1
⋮
  (collection_name attributes inherited from class ObjectA1)
⋮
CLASS NAME  **                      ObjectA2
⋮
  (collection_name attributes inherited from class ObjectA2)
⋮
⋮
CLASS NAME  **                      PartA 5
⋮
  (collection_name attributes inherited from parent class)
  PART  NAME  **********            PartB 6
⋮
END OF DISPLAY
BROW0410I 15.04.12 BROWSE OF COLLECTION COMPLETED
```

Figure 55. Collection Attributes Display Format

In our example, *collection_name* **1** is the name assigned to the collection. *collection_type* **2** is one of the collection types supported by TPFCS, such as ARRAY. Object part *PartA* **3** is the first part that is shown. The class hierarchy for this object begins with OBJECT **4**, which is the root class, and continues an inheritance hierarchy of *ObjectA1*, *ObjectA2*, and so on until the leaf class, *PartA* **5** itself, is reached. Notice that as discussed previously, every collection part object eventually inherits from the OBJECT class. The attributes of the OBJECT class consist solely of the TPFCS object header.

The attributes for each of these object classes are included in the display although they are not shown in this figure. After the *PartA* display, the beginning of the *PartB* **6** display is shown. Remember that *PartB* and *PartA* are not related in terms of inheritance, but are aggregate parts of *collection_name*. This display would continue in a similar way until all parts that comprise *collection_name* are included in the display.

Note: Data, key, and directory records are not part objects and are therefore not displayed by using the ZBROW COLLECTION command.

The following is an example of an actual display and is labeled with the same numbers as described in the previous example.

```
ZBROW COLLECTION ATTRIBUTES ARRAYCOL 1
BROW0602I 09.43.28 BROWSER QUALIFIED FOR DSNAME TO2SVTDS
BROW0406I 09.43.29 COLLECTION ATTRIBUTES DISPLAY
ATTRIBUTE                           VALUE
COLLECTION CLASS NAME - ARRAY 2
  PART  NAME  **********            OIDentry 3
CLASS NAME  **                      OBJECT 4
OBJECT_ID                           00000034
OBJECT_LGH                          0000028A
OBJECT_SEQ_CTR                      000000C8
CLASS NAME  **                      ObjectPart
OBJ_Part_CHANGE                     00
OBJ_Part_RESERVE2                   000000
OBJ_Part_PartID                     00000000
OBJ_Part_OIE                        00611748
CLASS NAME  **                      OIDentry 5
OIDentry_OID                        0200FC16 B06BF4DD E3D6F2E2 E5E3C4E2
****                                1800C28B 00000000 00000000 00000000
⋮
  PART  NAME  **********            ARRAY 6
CLASS NAME  **                      OBJECT
OBJECT_ID                           0000000C
OBJECT_LGH                          00000068
OBJECT_SEQ_CTR                      00000000
CLASS NAME  **                      ObjectPart
OBJ_Part_CHANGE                     00
OBJ_Part_RESERVE2                   000000
OBJ_Part_PartID                     00FE0001
OBJ_Part_OIE                        006071F8
CLASS NAME  **                      Collect
Collect_I_Struct_PartID             00FE0002
Collect_I_Struct_PTR                0060DBA8
Collect_I_Cursor                    00000000
Collect_I_DDEF_PartID               00FE0003
Collect_I_DDEF_PTR                  00603BE0
CLASS NAME  **                      ARRAY
ARRAY_I_NEXTINDEX                            201
ARRAY_I_MAXINDEX                              -1
⋮
  PART  NAME  **********            DASDFLATPool
⋮
  PART  NAME  **********            ObjectAccess
⋮
  PART  NAME  **********            DDEF_OBJ_DASD
⋮
DDEF_FORCE                          00
END OF DISPLAY
BROW0410I 09.48.54 BROWSE OF COLLECTION COMPLETED
```

Figure 56. Actual Collection Attributes Display

Not every attribute for these objects has been listed. You can determine what exact attributes each of these objects has, as well as the displacement and length of each attribute, by scanning the TPFCS source code for the CLASSC macro definition of each object.

See *TPF Operations* for more information and additional examples.

Determining Collection Residency

You can determine whether a collection is compact resident or extended resident by entering the ZBROW DISPLAY command with the COLLECTION parameter specified and examining the RESIDENCY TYPE field.

You can also determine the collection residency by displaying the parts of the collection. By examining the structure collection part for a collection, you can determine whether a collection is compact or extended as described in “Locating the Structure Object”.

Locating the Structure Object

One of the collection parts in a collection attributes display is the structure object used to represent the collection. (For extended-resident collections, there are also other objects that are located in pool file records chained to the structure object.) You can locate the file address of the structure object by using the ZBROW COLLECTION command with the ATTRIBUTES parameter specified to display the structure object attributes. You can find the file address of the pool record containing the structure object, otherwise known as the structure record, as follows:

- If the residency of the collection is compact, you will find the file address as part of the value indicated for the Structure_I_ID attribute.
- If the residency of the collection is extended, you will find the address as part of the directory entry, which is displayed for the StructDasd_I_Directory attribute.

As part of the response you receive from the ZBROW COLLECTION command with the ATTRIBUTES parameter specified, you will find either a CLASS NAME of StructureMem if the collection has a compact structure or a CLASS NAME of StructureDasd if the collection has an extended structure. As mentioned previously, all compact structures currently inherit from the StructureMem object class, whereas all extended structures currently inherit from the StructureDasd object class.

Locating the Data

The way that TPFCS stores the data for a collection on DASD depends on the structure object. As mentioned previously, for compact-resident collections, the data elements of the collection are stored in the structure object itself.[7] For extended-resident collections, the structure object is more complex and contains pointers to tree structures in the TPF database. The records that comprise these tree structures contain the data elements as well as control information for extended-resident collections.

Determining More Information about Pool File Records Used by TPFCS

One way to analyze a pool file record you suspect is being used by TPFCS to store a given collection in the TPF database is to do so manually using the ZDFIL command. While this procedure would be the most thorough, it requires a detailed knowledge of the information presented in this chapter, as well as a detailed knowledge of how each of the objects stored in the record is defined in the TPFCS source code. An easier approach is to use the ZBROW command with the FA parameter specified. This will provide some of the basic information about how that pool file record may [8] be used by TPFCS. If you need more detailed information, then use the ZDFIL command.

For more information about how to use the ZBROW command with the FA parameter, see *TPF Operations*.

---

[7] The structure object itself might not fit in a TPF 4-K record and, therefore, overflow into one or more additional records as we have discussed previously. If the structure object contains data, the data can be located in each of these records.

[8] The record might not ever be used by TPFCS even if it contains data that looks valid.

Scope of Current Validation and Reconstruction Support

The validation and reconstruction of collections currently provided by TPFCS by options on the ZBROW command can currently operate only on extended-resident collections. An error message results if you enter the ZBROW command with either the VALIDATE or RECONSTRUCT parameter specified for a compact-resident collection.

The following describes the scope of validation that TPFCS currently provides on extended-resident collections by using the ZBROW command:

- Validation currently does not perform automatic reconstruction or correction of errors reported, and reconstruction does not perform any preliminary validation of a collection.
- A subset and not all of the fields in the StructureDasd object are verified for those collections when validation is performed.
- The allocated and released data, key, and directory chains are verified when validation is performed.
- When validation is performed, an attempt is made to verify all directory entries, including those embedded in the StructureDasd object.
- Other than validation of directory entries, validation does not verify the contents of records on the data, key, and directory chains.
- Validation ends as soon as a major error occurs. As a result, a given validation report resulting in a response to a ZBROW command with the VALIDATION parameter might not indicate all of the errors present in a collection. It might be necessary for you to enter a validation request, followed by a reconstruction request or manual reconstruction, followed by another validation request. Depending on the nature of the errors found, you might have to use the ZBROW command several times, alternating between validation and reconstruction requests.

The following describes the scope of reconstruction that TPFCS currently provides on extended-resident collections by using the ZBROW command:

- Reconstruction does not perform any preliminary validation of a collection.
- The ZBROW command can only be used to perform one of the following types of reconstruction at any given time:
  - Reconstruction of the allocated data chain using information from key records located on the allocated key (index) chain (when present), and the directory entries located in the structure object as well as those located in the records on the allocated directory chain.
  - Reconstruction of the allocated key chain (when present) using information from the allocated data chain as well as the collection directory entries.
  - Reconstruction of both the allocated directory chain and the allocated key chain (when present) using information from the allocated data chain.

  Note: TPFCS can automatically rebuild even those allocated chains it uses to perform a given reconstruction operation before that operation has been completed.
- Reconstruction of the data chain normally does not change the contents of data records. Records are either placed on the new data chain or excluded from it.

For information about the ZBROW command, see *TPF Operations*.

Part 4. Coupling Facility Support

This part contains information about coupling facility (CF) support.

Coupling Facility Support

Coupling facility (CF) support provides data sharing capabilities that allow subsystems, system products, and applications running in a processor configuration to use a CF for high-performance, high-availability data sharing. The CF is a processor that attaches to other processors in a loosely coupled configuration to perform special-purpose operations. CF support provides connectivity to a CF for use by TPF system functions. CF support allows as many as 32 CFs and provides two types of *CF structures* that are used to perform various operations:

- *CF list structure*, which is a named piece of storage on a CF that enables the TPF system to share information organized as entries on a set of lists or queues. See “Coupling Facility List Structure” on page 235 for more information about CF list structures.
- *CF cache structure*, which is a named piece of storage on a CF that enables the TPF system to share information and allows high-performance sharing of frequently referenced data in logical entities. See “Coupling Facility Cache Structure Concepts” on page 241 for more information about CF cache structures.

You can add one or more CFs from a TPF processor to the processor configuration. Applications can connect to CF list or cache structures on CFs that have been added to the processor configuration and use connection services to manage data in that CF structure. See “Connection Services” on page 224 for more information. The first TPF system to connect to a CF structure allocates the structure in a CF and defines the structure attributes. When a TPF system no longer requires access to the CF structure, the TPF system can disconnect from that structure. See “Connecting to a Coupling Facility Structure” on page 230 and “Disconnecting from a Coupling Facility Structure” on page 234 for more information about connecting to and disconnecting from CF structures.

Other TPF systems can connect to the existing CF structure (either list or cache) by name, but cannot change the structure attributes of that CF structure as long as it remains allocated. See “Defining Structure Attributes for Coupling Facility Structures” on page 226 for more information about defining structure attributes.

An application that connects to a CF list structure can monitor individual lists to determine when list entries have been created on the list. When a list changes from empty state to nonempty state (that is, when a list entry is added to a previously empty list), an application-defined exit is called. This eliminates the need for application polling of lists and simplifies programming requirements.

An application that connects to a CF cache structure can automatically notify affected TPF systems when shared data in the cache is changed. The application can also determine whether the local copy of shared data is valid by checking system-maintained validity indicators.

Data Sharing Concepts and Terminology

Data sharing in a processor configuration refers to the ability of concurrent subsystems or applications to directly access and change the same data while maintaining data integrity and consistency throughout the processor configuration.

Throughout this chapter you will find the following terms used:

Term	Definition

Data	Any type of information; not only data contained in a database.
Application	Any subsystem, system product, or authorized application running on a TPF system in a multisystem environment or processor configuration. Typically, multiple instances of the application, distributed across the processor configuration, work together to perform a set of functions. For example, a database product could be installed on several systems in the processor configuration. On each system, an instance of the application accesses and manages the data that it shares with the other instances of the application.

Figure 57 shows a diagram of a CF with connected TPF systems.

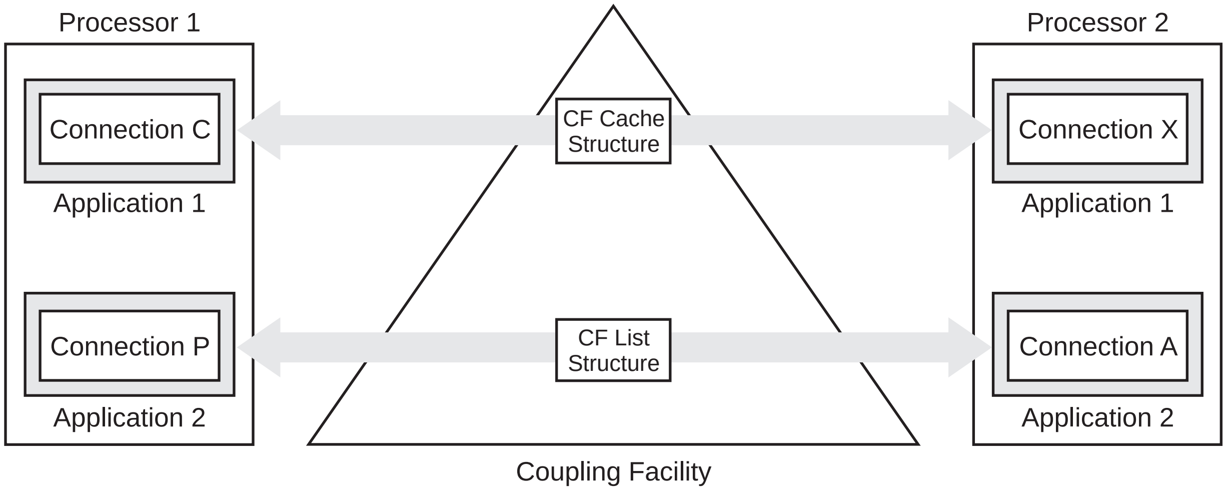


Figure 57. Multiple Systems Sharing Data through a CF

Connection Services

CF support provides services known as *connection services*, which allow authorized programs and subsystems to use the CF to share data in a processor configuration. The sections that follow discuss the connection services that manage connections to CF structures and include the following information:

- Connecting to a CF structure and causing allocation of the structure in a CF. To access connection services, you must first *connect* to a CF structure, specifying both a CF name and a structure type. The CFCONC macro allows you to connect to a CF structure. See “Connecting to a Coupling Facility Structure” on page 230 for more information.

  Note: You will need to connect again after TPF restart is completed.
- Disconnecting from a CF structure and causing deallocation of the structure in a CF. When you no longer need access to a CF structure, you can disconnect from that structure. The CFDISC macro allows you to do this disconnection. To access the CF structure at some later time, you must again connect to the structure using the CFCONC macro. See “Disconnecting from a Coupling Facility Structure” on page 234 for more information.

Other structure-specific information is made known to connectors through exit routines and, if applicable, you specify when you connect to a CF structure. The complete exit notifies you when a request that you submitted previously has completed. The list transition user notifies you when a list has changed from an empty state to a nonempty state. See “Defining Exit Routines” on page 251 for more information.

See the *OS/390 MVS Sysplex Services Guide* for more information about connection services.

Coupling Facility Commands

The commands are used to display entries in the coupling facility trace table (CFTT) and manage the CFs in a processor configuration. The following lists the CF commands and their functions:

Function	Description
ZCFCH	Manages CF cache structures.
ZDCFT	Displays entries in the CFTT. See “Coupling Facility Trace Table” on page 248 for more information.
ZMCFT ADD	Adds a CF to a processor configuration.
ZMCFT CLEAR	Removes from a CF the CF structures that are not known to this processor configuration.
ZMCFT DELETE	Removes a CF from a processor configuration.
ZMCFT DISPLAY	Displays the status of one or more CFs in the complex.
ZMCFT ENABLE	Resumes normal operations of a CF that was active previously, but became inactive when an error occurred.
ZMCFT REMOVE	Removes all inactive connections a processor has to a CF structure.
ZMCFT RESETLOCK	Resets a CF lock to an available state.

See *TPF Operations* for more information about these commands.

Using the Coupling Facility Commands

This section shows a scenario of how you might use the CF commands to manage CFs in a processor configuration by providing examples that span a series of commands.

Example 1: In the following example, two CFs are added to a processor configuration. First, a CF named CFONE, which is attached to symbolic device address (SDA) 1001, is added. Then, a CF named CFTWO, which is attached to SDA 510, is added.

A third CF named CFTHREE is enabled (normal operations are resumed on a CF that had been disabled) after a previous error interrupted it and the error has since been corrected. A TPF system tries to enable a fourth CF named CFFOUR after a previous error interrupted it and the error has ***not*** been corrected.

```
User:    ZMCFT ADD CFONE 1001
System:  MCFT0001I 07.59.28 CFMADD - COUPLING FACILITY CFONE ADDED - 2
                            PATHS EXIST

User:    ZMCFT ADD CFTWO 510
System:  MCFT0001I 07.55.28 CFMADD - COUPLING FACILITY CFTWO ADDED - 2
                            PATHS EXIST

User:    ZMCFT ENABLE CFTHREE
System:  MCFT0008I 07.52.15 CFMENA - COUPLING FACILITY CFTHREE ENABLED

User:    ZMCFT ENABLE CFFOUR
System:  MCFT0007E 07.50.15 CFMENA - COUPLING FACILITY CFFOUR COULD NOT
                            BE ENABLED
```

Example 2: In the following example, the fourth CF named CFFOUR is deleted from a processor configuration. By mistake, an attempt is made to delete the first CF named CFONE after a function started running on it.

```
User:    ZMCFT DELETE CFFOUR
System:  MCFT0002I 07.59.28 CFMDEL - COUPLING FACILITY CFFOUR DELETED

User:    ZMCFT DELETE CFONE
System:  MCFT0011T 07:57.24 CFMDEL - COUPLING FACILITY CFONE NOT DELETED - THERE
                            ARE STILL STRUCTURES ALLOCATED
```

Example 3: In the following example, you tried to add the first CF named CFONE to a processor configuration again. You also tried to add a fifth CF named CFFIVE to a processor configuration, but the SDA you specified is already being used by another CF.

```
User:    ZMCFT ADD CFONE 500
System:  MCFT0006T 07.59.20 CFMADD - COUPLING FACILITY CFONE HAS ALREADY BEEN
                                     ADDED

User:    ZMCFT ADD CFFIVE 510
System:  MCFT0020T 07.59.25 CFMADD - COUPLING FACILITY ATTACHED TO SDA 510 HAS
                                     ALREADY BEEN ADDED WITH NAME CFTWO
```

Coupling Facility Structure Concepts

Whether a CF structure is defined as a CF list structure or a CF cache structure, certain characteristics are common to both types. The topics that follow provide basic information about both types of CF structures.

Defining Structure Attributes for Coupling Facility Structures

When using the CFCONC macro to connect to a CF structure, specify structure attributes that describe the CF structure you need. Whether the TPF system uses the structure attributes you specify depends not only on your CFCONC parameters, but also on resource availability in the CF and whether you are the first to issue the CFCONC macro for the CF structure, therefore causing its allocation.

The CF structure to which you receive connectivity may or may not meet all your requirements. The TPF system returns the actual structure attributes to you in the CFCONC answer area, which is mapped by the ICFAA data macro. It is your responsibility to verify that the structure attributes, as indicated in the CFCONC answer area, are acceptable. If you decide not to accept one or more of the structure attributes, you can disconnect from the CF structure.

See “Specifying Structure Attributes for Coupling Facility Structures” on page 231 for a description of the structure attributes that are required on the CFCONC macro.

Identifying Connection States

A connection to a CF structure can be in one of two states:

- *Undefined state*, where the connection does not exist
- *Active state*, where the connection is active.

Understanding Structure Persistence

The structure attribute of persistence applies to CF list structures and CF cache structures. The persistence attribute of a CF structure is affected by how you define your structure disposition.

The structure disposition (the STRDISP parameter on the CFCONC macro) determines whether the CF structure remains allocated when there are no active connections to either CF structure. A structure disposition of KEEP indicates that when there are no active connections to the CF structure, that structure remains allocated. For example, if data in the CF structure needs to be kept permanently in the CF, specify a structure disposition of KEEP by coding STRDISP=KEEP on the CFCONC macro. A CF structure that remains allocated when there are no active connections is called a *persistent structure*.

A structure disposition of DELETE (by coding STRDISP=DELETE) indicates that when there are no active connections to the CF structure, that structure is deallocated. However, if there any active connections to the CF structure, that structure remains allocated.

See “Specifying Structure Attributes for Coupling Facility Structures” on page 231 for more information about the STRDISP parameter. See *TPF System Macros* for more information about the CFCONC macro.

Allocating a Coupling Facility Structure

The allocation of a CF structure depends on the following factors:

- Application requirements
- Availability of CF storage.

The request of the application for CF structure allocation, combined with the storage usage requirements of the control code, ultimately determines if, where, and how large a CF structure is allocated.

TPF System Considerations

When the CF structure is allocated successfully, the TPF system sets a return code of zero and the ICFCAACONNALLOC bit in the CFCONC answer area is set to indicate that the connection caused the CF structure to be allocated. However, you still must verify that the structure attributes finally assigned to the CF structure are acceptable.

The CFCONC answer area for a CF structure contains:

- The vector length that the TPF system obtained (the ICFCAAVECLEN field). This length may be less than what you requested on the CFCONC macro.
- Control information for the CF structure, such as the maximum number of data elements for each list entry (the ICFCAALISTMAXELEMNUM field).

Coupling Facility Structure Size

The requested size of the CF structure is specified on the CFCONC macro using the STRSIZE parameter. The actual size of the CF structure allocated is returned in the ICFCAASTRSIZE field of the CFCONC answer area. See the *S/390 Processor Resource/Systems Manager Planning Guide* for more information about calculating the size of the CF structure. See "Specifying Structure Attributes for Coupling Facility Structures" on page 231 for more information about the STRSIZE parameter. See *TPF System Macros* for more information about the CFCONC macro.

The CF ensures that the size of a CF structure is a multiple of the CF storage increment. See "Coupling Facility Storage Increment" on page 229 for more information. If not, the CF rounds up the size value to be a multiple of the increment. Ultimately, the actual size of the CF structure allocated is based on storage allocation priorities that comply with the CF control code and on storage constraints in the CF itself. See "Coupling Facility Considerations" for more information about these CF allocation considerations.

Coupling Facility Considerations

CF structure size includes both control areas required by the CF control code and data areas used by the application. The size is also affected by CF allocation rules and the CF allocation increment size, which is a function of the CF level.

You must take all these factors into consideration when determining how to configure your CF.

Coupling Facility Storage

There are two types of storage in a CF:

- Control storage
- Noncontrol storage.

The CF control code uses each type of storage for a specific purpose. Essentially, the CF control code uses*control storage* either for its control information or for data, and the *noncontrol storage* only for data. Depending on the particular processor on which the CF is defined, the storage can be all control storage or a combination of control and noncontrol storage. You control the amount of storage assigned to control and noncontrol storage when you configure the amount of central and expanded storage in the CF. The amount of central storage equates to the amount of control storage; the amount of expanded storage equates to the amount of noncontrol storage. In processors that do not support the concept of central and expanded storage, all CF storage is considered to be control storage.

The division of control storage and noncontrol storage becomes a consideration when the CF control code allocates specific amounts of storage to a CF structure. The division between the two storage types must be considered to ensure that the CF storage is distributed most effectively for use by the application.

Coupling Facility Resource Allocation Rules

A CF structure is allocated at a certain size based on characteristics of the CF itself, such as storage constraints and the storage increment.

Coupling Facility Storage Constraints

The CF control code locates parts of a CF structure in either control storage or noncontrol storage depending on whether the part is control information or data.

Control information ***must*** reside in control storage and cannot reside in noncontrol storage; data may reside in either control storage or noncontrol storage.

The amount of control storage available can affect the allocation of CF structures. When there is no control storage available in the CF, control information, such as list entry controls or directory entries, cannot be allocated (even though there may be ample available noncontrol storage in the CF structure).

The following summarizes the actions taken when there is no more available control storage in a CF structure:

- Entries, which are control information, cannot be allocated because they must reside in control storage.
- If data elements are requested in the CF structure, they can continue to be allocated because they can reside in noncontrol storage. This action causes the actual entry-to-element ratio to be changed in favor of elements.
- The CF control code continues allocating data elements until one of the following occurs:
  - The total space for the CF structure is equal to the requested total structure size.
  - The number of elements allocated equals the number of elements that result in an actual achieved entry-to-element ratio of 1 divided by the value specified for the MAXELEMNUM parameter on the CFCONC macro.

    Note: The value specified for the MAXELEMNUM parameter is specified by the application when it connects to the CF structure and indicates the maximum number of data elements for each list entry that are supported for any list entry in the CF structure.

    Even though the total requested CF structure size may not be reached when the ratio of 1 divided by the MAXELEMNUM parameter value is reached, additional data elements are not allocated.

Coupling Facility Storage Increment

CF storage is allocated in multiples of the CF model-dependent storage increment size. For CF level 6, all CF structure allocations are rounded up to a multiple of 256 KB.

Successfully Completing Coupling Facility Structure Allocation

Each time you successfully issue the CFCONC macro for a CF structure, the TPF system places a connect token in the CFCONC answer area. The *connect token* identifies each connection to the CF structure and is unique for a connection in the processor configuration. You can issue the CFCONC macro from any TPF system in the processor configuration that is connected to the CF.

Figure 58 shows task 1 allocating a CF structure for the first time.

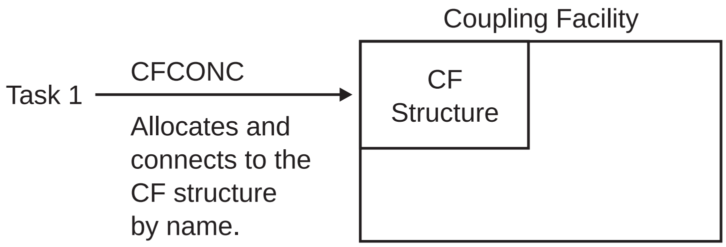


Figure 58. Allocating a CF Structure

In Figure 59, task 2 connects to the same CF structure.

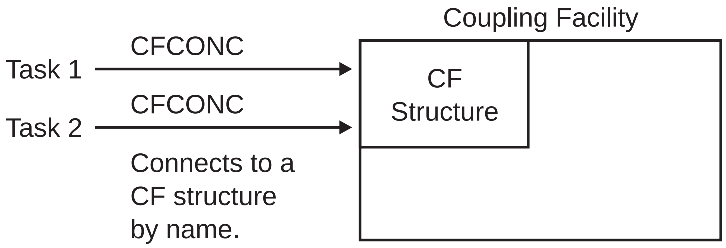


Figure 59. Connecting to an Allocated CF Structure

Each connector, whether it is the first connector or a subsequent connector to a CF structure, must verify that the structure attributes are acceptable.

- For the first connector, even though the return code may be zero (0), the CFCONC macro may not have satisfied all structure attributes requested. The CONACONNALLOC flag is set to indicate that this connection allocated the CF structure in the CF.
- Subsequent connectors to a CF structure that is allocated already must verify that the structure attributes established by the first connector to the CF structure or received in the CFCONC answer area are acceptable.

If you find that the structure attributes are not acceptable, you can disconnect from the CF structure using the CFDISC macro. See “Disconnecting from a Coupling Facility Structure” on page 234 for more information.

Connecting to a Coupling Facility Structure

You connect to a CF structure to use connection services to manage data in that CF structure. See “Connection Services” on page 224 for more information about connection services.

Overview of Connect Processing

You connect to a CF structure to manage structure data. The first TPF system to connect to a CF structure causes the structure to be allocated in a CF using the structure attributes specified on the CFCONC macro. Subsequent connectors to the CF structure cannot change those initial structure attributes. The number of TPF systems that are allowed to connect to a CF structure is a function of the CF model.

Once your CFCONC request is completed successfully:

- You can receive data in the CFCONC answer area, which is mapped by the ICFCAA DSECT.
- You are connected to the CF structure you requested.

- You can request structure services that are valid for a CF structure.

Specifying Structure Attributes for Coupling Facility Structures

The following CFCONC parameters define the structure attributes for the CF structure. See the CFCONC macro in *TPF System Macros* for detailed information about each parameter.

Parameters Common to Both Structure Types

Parameter	**Description**
CFLEVEL	Use this parameter to specify the level of the CF in which the CF structure will be allocated. If you try to connect with a CF level higher than that of the CF, your CFCONC request fails with return code ICFRRCBADCFLEVEL. The maximum CF level value supported is returned in the CFCONC answer area in the ICFCAATPFMAXCFLEV field. The CF level in which the CF structure is actually allocated is returned in the ICFCAACFLEVEL field. The CF level can be specified only through the CFCONC macro. To change to a different CF level, you must disconnect and then connect to the CF structure again with a different value. See “Coupling Facility Considerations” on page 228 and “Disconnecting from a Coupling Facility Structure” on page 234 for more information.
CFNAME	Use this parameter to specify the CF name, which is needed to allocate the CF structure.
CONDATA	Use this parameter to provide 8 bytes of connect data. The CF passes this data to your exit routines when called and is for your use only. A possible use for this parameter is as a pointer to a control block that represents the connector.
CONNAME	Use this parameter to identify your connection to the CF structure.
STRNAME	Use this parameter to name the CF structure to which you want to connect. You must supply this name to users of your application and then use the TYPE parameter to indicate that the structure you want allocated is a CF list structure or a CF cache structure.
STRSIZE	Use this parameter to specify the size of the CF structure in 4-KB blocks.
STRDISP	Use this parameter to specify the disposition of the CF structure when all connections are released.
TYPE	Use this parameter to identify the type of CF structure in the CF to which you want to connect. For a CF list structure, the type is LIST. For a CF cache structure, the type is CACHE.

Parameters for CF List Structures

Parameter	Description
ADJUNCT	Use this parameter to specify whether the CF list structure will contain adjunct data areas.
ELEMCHAR	Use this parameter to specify the data element size to be used.
ELEMENTRATIO	Use this parameter to specify the element component of the entry-to-element ratio.
ELEMINCRNUM	Use this parameter to specify the data element size to be used.
ENTRYRATIO	Use this parameter to specify the list entry portion of the entry-to-element ratio.
LISTCNTLTYPE	Use this parameter to specify whether the amount of CF storage that may reside on a given list header is to be controlled by limiting the maximum number of entries or the maximum number of data elements.
LISTHEADERS	Use this parameter to specify the number of lists to be allocated in the CF list structure.
LISTTRANEXIT	Use this parameter if you plan to use list monitoring because the parameter specifies the address of your list transition exit routine.
LOCKENTRIES	Use this parameter to specify the number of lock entries for a serialized CF list structure.
MAXELEMNUM	Use this parameter to specify the maximum number of data elements for each data entry.
REFOPTION	Use this parameter to specify whether list entries are to be referenced by entry name, entry key, or neither. List entries can always be referenced by the list entry identifier (LEID) or unkeyed position.
VECTORLEN	Use this parameter if you plan to use list monitoring because the parameter specifies the maximum number of list headers that you can monitor for transitions between empty state and nonempty state.

Parameter for CF Cache Structures

Parameter	Description
VECTORLEN	Use this parameter to specify the number of cache buffers in the local storage of the requester that require concurrent registration.

Determining Whether a Connection Is Successful

When you issue the CFCONC macro, you identify the storage area (using the ANSAREA parameter) where the TPF system returns information about the success or failure of your connect request. Note the following about the return codes shown in Table 18 on page 233 and Table 19 on page 233:

- All five return codes are returned in the high-order 2 bytes of register 15 (R15).

- If the return code is not zero, the low-order 2 bytes of R15 contain reason codes. See *TPF System Macros* for an explanation of these reason codes.

If your request to connect to a CF structure is successful, one of the following return codes, as shown in Table 18, is returned.

Table 18. Return Codes for a Successful Connection to a CF Structure

Return Code	Equate Symbol	Description
0000	ICFRRCOK	Your connection is successful. The TPF system has returned data to you in the CFCONC answer area.
0004	ICFRRCWARNING	Your connection is successful, but you may need to do additional processing based on the information returned to you in the CFCONC answer area.

If your request to connect to a CF structure is ***not*** successful, one of the following return codes, as shown in Table 19, is returned in R15.

Table 19. Return Codes for an Unsuccessful Connection to a CF Structure

Return Code	Equate Symbol	Description
0008	ICFRRCPARMERROR	You have incorrectly specified a parameter on your CFCONC request.
000C	ICFRRCENVERROR	There is an environmental error.
0010	ICFRRCCOMPONENT	A system failure occurred.

See “Receiving Information in the CFCONC Answer Area” for more information.

Receiving Information in the CFCONC Answer Area

When you issue the CFCONC macro, you identify the storage area where the TPF system will return information about the status of your request. Use the following parameters to specify this CFCONC answer area:

Parameter	Description
ANSAREA	Contains the address of the CFCONC answer area. Use the ICFCAA DSECT to map the CFCONC answer area.
ANSLEN	Contains the length of the CFCONC answer area. The length must be large enough to hold the CFCONC answer area mapped by the ICFCAA DSECT.

Successfully Completing a Connection

The CFCONC macro returns information about a successful connection in the ANSAREA area. See the ICFCAA DSECT for information about the successful completion of a connection.

Handling Failed Attempts to Connect to a Coupling Facility Structure

When a connection request is not successful, you must consider the conditions that could have caused the rejection. In a short-term condition, consider issuing the connect request again as soon as possible.

Another type of condition may require your intervention and, therefore, take a significantly greater amount of time to resolve. It may be necessary to reconfigure connectivity to a CF. The following are examples of longer-term conditions that cause a connect request to fail:

- All connections to a specific CF structure are in use.
- The requesting TPF system does not have connectivity to the CF that contains the specified CF structure.
- The CF function is not active; for example, CF restart did not end successfully or the TPF system has not allocated the fixed file records.

Disconnecting from a Coupling Facility Structure

A connected TPF system can disconnect from a CF structure when you no longer require access to that structure. Once disconnected, the TPF system cannot access the CF structure through any connection services.

TPF systems disconnect from a CF structure either for normal processing or because of a failure. The connect token is invalidated for the disconnecting TPF system.

Disconnection Parameters for the Coupling Facility Structure

The CFDISC macro allows you to disconnect from a CF structure when you no longer require access to it. You can disconnect from only one CF structure at a time. If you want to disconnect from multiple CF structures, issue the CFDISC macro once for each CF structure.

The CFDISC macro requires you to provide the connect token on the CONTOKEN parameter that was returned by the connection services when the initial connection to the CF structure was made with the CFCONC macro.

See *TPF System Macros* for more information about the CFDISC macro.

Persistence Considerations

Structure persistence is defined at connect time using the STRDISP parameter of the CFCONC macro. Coding STRDISP=KEEP and STRDISP=DELETE indicates whether a CF structure will become undefined after all TPF systems have disconnected from it. See “Specifying Structure Attributes for Coupling Facility Structures” on page 231 for more information about the STRDISP parameter. See *TPF System Macros* for more information about the CFCONC macro.

The TPF system releases the connection to the CF structure when a normal disconnect occurs. You can also disconnect from the CF structure in an error recovery condition.

Handling Resources for a Disconnection

After all active TPF systems have disconnected from the CF structure, connection services either deletes or retains the CF structure depending on the structure disposition you specified for the STRDISP parameter of the CFCONC macro. See “Defining Structure Attributes for Coupling Facility Structures” on page 226 for more information.

Whether the disconnect is normal or the result of an error, the TPF system cleans up resources for a CF structure. For example, the list monitoring interest, if registered, is released.

Successfully Completing a Disconnection

The connect token for the TPF system is invalidated before returning control to the TPF system that issued the CFDISC macro. This ensures that the TPF system cannot issue additional connection services mainline requests. If the TPF system does use the invalidated connect token to issue a request for connection services, the request fails.

Coupling Facility List Structure Concepts

The topics that follow topics provide basic information about the CF list structure.

See "Allocating a Coupling Facility Structure" on page 227 for more information about how the TPF system handles a request for CF list structure allocation. Understanding this concept will help you provide assistance to anyone using CF support.

Coupling Facility List Structure

Rather than accessing data in a CF by address, you can allocate objects called *structures* and access data in the structures as logical entities (for example, by name). The ability to access data this way frees you from having to be concerned with the physical location or address of the data. One type of object that CF support provides is a *CF list structure*. Each CF list structure must reside entirely in a single CF and applications can use multiple CF list structures.

The CF can support multiple occurrences of the CF list structure at the same time. However, a CF list structure found in one CF cannot be seen by another CF. A CF list structure consists of a number of lists, each with a number of list items.

The characteristics and services associated with a CF list structure support certain types of uses and offer certain unique functions. A *CF list structure* is a named piece of storage on a CF that enables TPF systems to share information organized as entries on a set of lists or queues. Connections could use a CF list structure, for example, to distribute work or maintain shared status information. By using a CF list structure, you can monitor list transitions from an empty state to a nonempty state without accessing the CF and checking the lists directly.

A CF list structure consists of a set of lists and an optional lock table of exclusive locks that you can use to serialize the use of lists, list entries, or other resources in the CF list structure. A list header points to a CF list, which can contain a number of list entries. A *list entry* consists of list entry controls and can also include a data entry, an adjunct area, or both. Data entries and adjunct areas are both optional. However, data entries are optional for each list entry while adjunct areas exist for all list entries or no list entries. A CF list structure that includes a lock table is called a *serialized list structure*. Figure 60 on page 236 shows a serialized list structure.

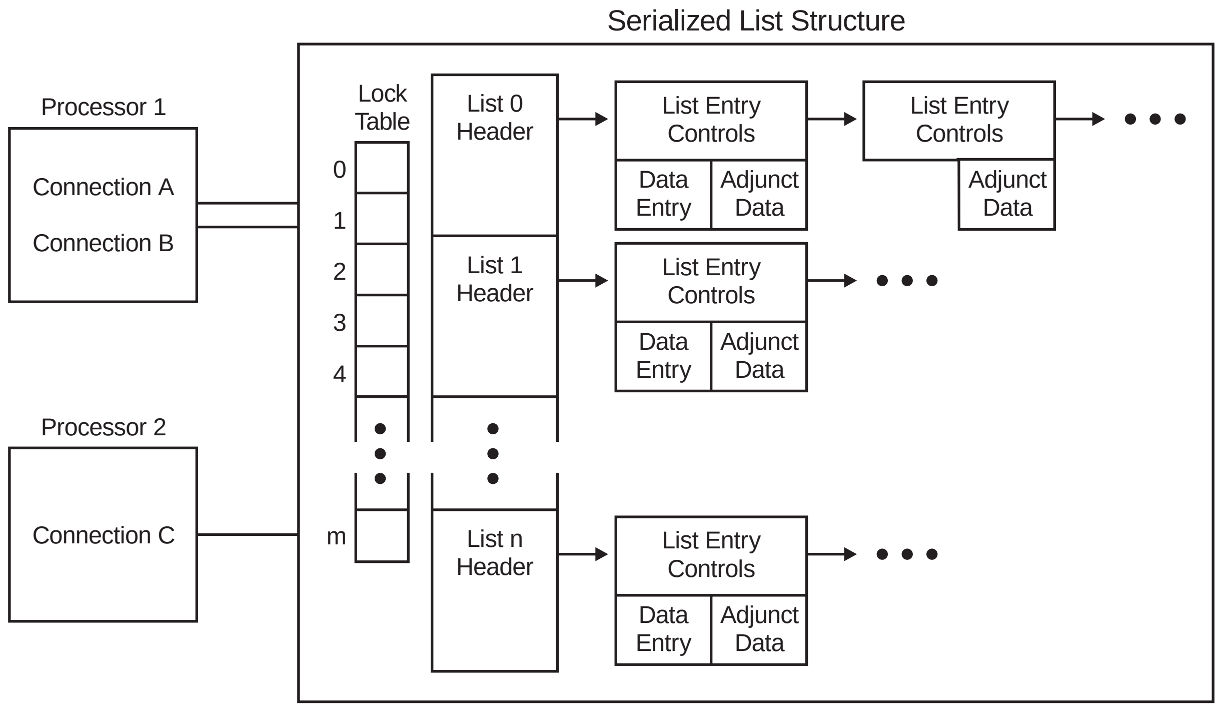


Figure 60. A Serialized List Structure

The following are the parts of the CF list structure:

Part	**Description**
List Header	Anchors the list to the CF list structure and contains control information that is associated with the list. The control information is known as *list controls*. The first TPF system to connect to the CF list structure designates the number of list headers it is to have and allocates the CF list structure.
List Entry	An entry on the list. Data in the CF list structure is stored in list entries, each of which can consist of a data entry of up to 16 data elements in a CF level 6 and an optional adjunct data area. A list entry contains the following: • List entry control, which contains control information associated with the list entry. • An optional data entry that holds user-defined data. Data entries contain units of storage called *data elements*. For CF level 6, data entries can contain 0 to 255 data elements and a data entry can contain up to 64 KB (65 536 bytes) of data. • An adjunct area that is used to hold up to 64 bytes of data. You can use the adjunct area to maintain control information about the contents of the data entry. If your data is always 64 bytes or less, you can use the adjunct area to hold your data and omit the use of data entries. Each list entry can reside on only one list at a time. Unused list entries do not reside on any list.
Lock Table	An array of exclusive locks that you can use to serialize access to CF list structure resources such as lists or list entries. The purpose and scope of the exclusive locks are defined by the application.

Lock table users create and maintain the association between a lock table entry and its associated resource. The lock table can be used:

- Together with list entry operations such as reading or writing list entry data
- Independently of list entry operations.

See “Coupling Facility Locking Functions” on page 249 for more information about a serialized list structure, the CF locking functions, and the lock table.

How Data Is Maintained in a Coupling Facility List Structure

Data in the CF list structure is stored in list entries, each of which can consist of a data entry up to 255 data elements in a CF level 6 and an optional adjunct data area. The number of data element sizes and the range in the number of elements for each data entry provides a tremendous choice of data entry sizes. The maximum data entry size is 64 KB, except in a CF list structure that has a data element size of 256 bytes. Because the maximum number of data elements for each data entry is 255, the maximum data entry size with 256-byte data elements is 65 280 bytes (255 × 256). All other combinations of data element size and data entry size allow a maximum of 64 KB (65 536 bytes). Although a data entry consists of a number of data elements, list operations handle the data entry as a single entity; data elements cannot be read or written individually. The adjunct area can be used to hold additional, user-defined information about the data entry.

Figure 61 shows a list that contains list entries with various numbers of data elements; list entry controls are not shown in this figure. See the *OS/390 MVS Sysplex Services Guide* for more information about how data is maintained in a CF list structure.

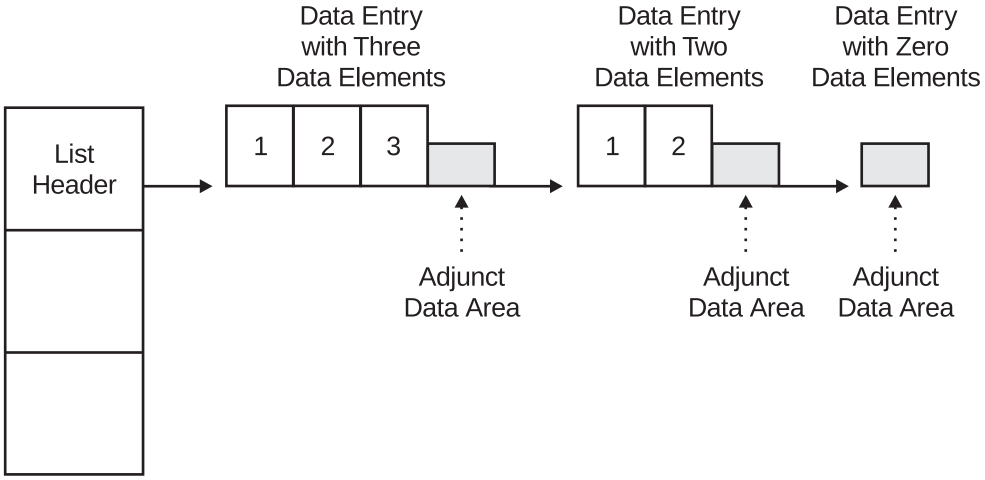

Data Entry with Three Data Elements
Data Entry with Two Data Elements
Data Entry with Zero Data Elements
List Header
1
2
3
1
2
Adjunct Data Area
Adjunct Data Area
Adjunct Data Area

Figure 61. A List Containing Entries with Various Numbers of Data Elements

Specifying Connection Parameters for the Coupling Facility List Structure

This section provides general information about how to code the CFCONC macro so you can connect to a CF list structure. See *TPF System Macros* for detailed information about each parameter of the CFCONC macro.

Data Element Size

To select the data element size for a CF list structure you must understand the approximate sizes of the smallest and largest pieces of data you want stored in the list entries. If the data can fit into adjunct areas, you can avoid using data entries altogether. If you specify 0, the CF list structure is allocated without data elements. The TPF system ignores the MAXELEMNUM parameter if you specify it with the ELEMENTRATIO parameter set to 0.

The CF allows a maximum of 255 data elements for each data entry, but you can use the MAXELEMNUM parameter to specify a smaller maximum number if you want to additionally restrict the size of the largest data entries.

The value you specify for the MAXELEMNUM parameter must be greater than or equal to the value specified for the ELEMENTRATIO parameter divided by the value specified for the ENTRYRATIO parameter:

MAXELEMNUM >= (ELEMENTRATIO / ENTRYRATIO)

The data element size multiplied by the maximum number of data elements must be enough to accommodate the largest piece of data that you need to manage as a single entry.

Entry-to-Element Ratio

You cannot directly control the number of list entries or data elements the CF list structure holds. When the CF list structure is allocated, its storage is subdivided to reserve space for CF list structure components like data elements and list entry controls. The value you specify for the entry-to-element ratio is used by the TPF system to determine the proportion of the CF list structure storage to allocate to each component. The ratio, expressed as a pair of whole numbers like 1:4, is passed to the CFCONC macro using the ENTRYRATIO and ELEMENTRATIO parameters:

- The ENTRYRATIO parameter specifies the part of the ratio for the list entries; for example, the 1 in the 1:4 ratio.
- The ELEMENTRATIO parameter specifies the part of the ratio for the data elements; for example, the 4 in the 1:4 ratio.

In general, the entry-to-element ratio should reflect the average number of data elements for each list entry. For example, if your data element size is 4096 bytes and you estimate that about half of the list entries will require 1 data element and about half of the list entries will require 8 data elements, you want a ratio of 1:4.5, which you would express in whole numbers as 2:9.

Although you request a particular entry-to-element ratio through the CFCONC macro, the CF may use a slightly different ratio. The actual number of entries and elements in the CF list structure, rather than the ratio, is returned to you in the CFCONC answer area, which is mapped by the ICFCAA DSECT.

Note: The values in the CFCONC answer area are ***not*** exact values.

Limiting the Storage Used by Each List

The LISTCNTLTYPE parameter allows you to choose how storage use is managed for individual lists. You can limit either the number of list entries for each list or the number of data elements for each list. A limit on storage use for each list may be needed to prevent the excessive use of storage by certain lists.

The flexibility offered by the choice of limits allows you to select the type of limit that best suits your use of the CF list structure. For example, if your main concern is to limit the number of entries that may build up on a list, limit the number of list entries for each list. If your main concern is to prevent the entries on a given list from consuming too much of the storage in a CF list structure, limit the number of data elements on a list.

Adjunct Areas

The adjunct area can contain 64 bytes of user-defined data such as information about the status of the data entry or time stamp. The adjunct area is maintained separately from the data entry so you can change the contents of the data entry or the adjunct area independently.

Named or Keyed List Entries

Named entries let users reference list entries by a user-defined name. Keyed entries let users maintain list entries in a keyed order. The choice of named or keyed entries, or the use of neither, depends on how you are using the CF list structure. For example, if the list entries represent units of work ordered by priority, you may choose keyed entries. If the list entries represent customer records in a particular category, you may choose named entries. If the lists represent units of work to be processed on a first-in-first-out (FIFO) basis, there may be no need for names or keys. Use the REFOPTION parameter on the CFCONC macro to specify how to reference the list entries in the CF list structure.

List Transition Exit

CF support provides the list transition exit routine, which plays a critical role in the operation of the CF list structure. TPF systems provide the address of the exit routine when they issue the CFCONC macro to connect to the CF list structure. See “Defining Exit Routines” on page 251 for more information.

Checking or Modifying a List Notification Vector

The CFVCTC macro allows you to perform the following functions on a list notification vector associated with a CF list structure:

- Test whether a list is empty or not empty
- Test a range of list notification vector entries to determine whether the associated lists are empty or not empty
- Modify the number of entries in a list notification vector.

List Notification Vector

When you issue the CFVCTC macro, you identify the list notification vector using the vector token returned in the CFCONC answer area (mapped by the ICFCAAVECTOK field of the CFCONC macro) when you issued the CFCONC macro to connect to the CF list structure. You receive a vector token from the CFCONC macro only if you coded the VECTORLEN parameter.

Changing the Number of Entries in a List Notification Vector

Coding REQUEST=MODIFYVECTORSIZE on the CFVCTC macro allows you to change the number of entries in the list notification vector so that you can monitor a different number of lists in the CF list structure.

Reducing the size of the list notification vector when it is larger than necessary frees storage for the list notification vectors of other users in the TPF system. Use the VECTORLEN parameter to indicate the new number of entries you would like the list notification vector to contain. If the value you specify is not a multiple of 32, the TPF system rounds up the value to a multiple of 32.

The number of entries the TPF system actually assigns to the list notification vector is returned to you as output through the ACTUALVECLEN parameter.

Decreasing the Number of Entries: If you request a decrease in the number of entries in the list notification vector, your request will always be satisfied. When the size of a list notification vector is decreased, the number of entries is reduced by removing entries starting with the highest number. The remaining entries are unchanged and retain their original values (empty state or nonempty state).

Before eliminating any entries, you must ensure that the entries that will be deleted are not being used to monitor lists.

If multiple TPF systems could be accessing list notification vector entries at the same time, you should obtain exclusive serialized access to the list notification vector before decreasing its size. Otherwise, if you code REQUEST=LTVECENTRIES or REQUEST=TESTLISTSTATE must be prepared to handle the ICFRRCINVALIDINDEX return code, which indicates that the specified vector index is no longer valid.

Increasing the Number of Entries: If you request an increase in the number of entries in the list notification vector and the TPF system is unable to obtain enough storage to satisfy your request, the new number of entries may be unchanged or smaller than you requested. When this occurs, the number of entries returned through the ACTUALVECLEN parameter will be smaller than the requested number and you will receive the ICFRRCLESSTHAN return code to inform you of the result.

When the size of the list notification vector is increased, the number of entries is increased by adding additional entries after the current highest-numbered entry. Existing entries are unchanged and retain their original values (empty state or nonempty state). New entries are initialized to the nonempty state.

Testing Whether a List Is Empty

Coding REQUEST=TESTLISTSTATE on the CFVCTC macro allows you to test the entry representing a particular list to determine whether that list is empty. List notification vector updates are performed asynchronously by the TPF system, so a list notification vector entry may not show a particular list state change at the time you check it. However, the CF architecture ensures that the change in the list notification vector will be performed. The CF architecture also ensures that if the list transitions from an empty state to a nonempty state and then back to an empty state and so on multiple times, the final state reflected in the vector will match the final state of the list. However, individual transitions may not be applied to the vector if they are superseded by subsequent changes. For example, if the initial state of the list notification vector entry indicates that the list is empty and then the list transitions to a nonempty state and becomes empty again in a short period of time, the TPF system does not ensure that the interim nonempty state will be reflected in the vector. However, the TPF system ensures that the final state (an empty state, in this example) is correct.

Testing Whether a Range of Lists Is Empty

Coding REQUEST=LTVECENTRIES on the CFVCTC macro allows you to test as many as 32 consecutive list notification vector entries to determine whether their associated lists are empty. The output from this request is a bit string with 1 bit for each list notification vector entry, starting with the vector entry you specify as the starting vector entry number and continuing until 32 bits are loaded. List notification vector entries range from 0 to $n-1$, where n is the number of entries in the list notification vector.

The bit values in the bit string are interpreted as follows:

Bit Value	Interpretation
0	The bit value indicates that the corresponding monitored list is not empty.
1	The bit value indicates that the corresponding monitored list is empty.

Coupling Facility Cache Structure Concepts

A CF cache structure allows high-performance sharing of frequently referenced data in logical entities and provides you with data consistency and high-speed access to data. Data consistency means that you can develop protocols to ensure the data that they share is valid. High-speed access means that you can develop data sharing programs and protocols with improved performance. You can do the following:

- Automatically notify TPF systems when the shared data is changed. The TPF system keeps track of who is using a particular piece of data and notifies them when an update to the data makes their locally cached version obsolete.
- Determine when your copy of shared data is valid by checking system-maintained validity indicators for your locally cached copies of shared data.

The TPF system supports directory-only CF cache structures. See “Elements of a Cache System” on page 242 for more information about directory-only CF cache structures. See “Benefits of Using Coupling Facility Cache Structures” on page 242 for more information about data consistency and high-speed access to shared data.

Terminology

The following lists terms that describe the basic concepts that are important to understand the CF cache structure.

deregistration/deregistering interest	A way to indicate to you information about the validity of a piece of shared data. If you have a *registered interest* in a piece of shared data, you can have your interest *deregistered* if that piece of shared data has changed and the local copy of the data is no longer valid. When a piece of shared data is updated, the TPF system indicates to those interested, through the associated local cache vector entry, that the piece of shared data has been changed. The copy of the data in the local cache buffer is then considered to be *not valid*. This process is also referred to as *invalidation* of local cache copies of pieces of shared data.
directory-only cache	A CF cache structure that contains directory entries but not pieces of shared data. See “Elements of a Coupling Facility Cache Structure” on page 244 for more information about a directory-only cache.
invalidation	See *deregistration/deregistering interest*.
registration/registering interest	A way to indicate to you information about the validity of a piece of shared data. If you use the CF cache structure, you can *register* interest in a piece of shared data. When you register interest in a piece of shared data, an association is formed between the local cache vector entry associated with

your local copy of the data and the directory entry for the data in the CF cache structure. When interest has been registered, the TPF system uses the local cache vector entry to indicate whether the data in your local cache buffer is valid. If you have registered interest in a piece of shared data, the copy of that data in your local cache buffer is considered to be *valid*.

valid data
The state of data in your local cache buffer. If your copy of a piece of shared data is *valid*, the copy contains the latest changes. If a copy of the data is *not valid*, it does not reflect the latest changes. See also *registration/registering interest*.

validation
See *registration/registering interest*.

Benefits of Using Coupling Facility Cache Structures

This section discusses some benefits of using CF cache structures.

Data Consistency

You can use the CF cache structure to keep track of data that resides in the local cache and in permanent storage. No matter how you store data that multiple TPF systems share, each CF cache structure is expected to maintain a local cache buffer to contain a *copy* of the data. By using a directory in the CF cache structure and a mechanism called *cross-invalidate* to inform TPF systems of changes to data, each TPF system in the complex can keep track of whether locally cached copies of the data are valid; that is, whether the copies contain the latest changes.

The directory allows you to refer to named data items that you can store in local storage. Cross-invalidate processing involves setting an indicator in a local cache vector for each of the TPF systems to indicate whether the locally cached copy of the data is valid. TPF systems must test the indicator to determine if their copy is valid and, if the data is no longer valid, they must read the data from permanent storage to obtain the most current copy.

High-Speed Access to Shared Data

You can use the CF cache structure to keep track of shared data that TPF systems maintain in their local cache buffers. Accessing data stored in the local cache buffer is the quickest way for a TPF system to access the shared data. However, if the TPF system has invalidated the local copy because another system has updated the data, the TPF system must gain access to the data from permanent storage.

See *TPF Application Programming* for more information about local cache buffers and permanent storage.

Elements of a Cache System

A cache system contains the following elements:

- *CF cache structure*, which is a structure in the CF that contains a directory to keep track of data that is shared among cache users. TPF systems that are connected to the CF cache structure can manage shared data.
- *Permanent storage*, which is storage that is the final repository for the data that TPF systems share and might be on DASD. TPF systems can read the data from permanent storage to local cache buffers and use the directory-only caching method to track the validity of the data. After TPF systems make updates to the locally cached data, they are responsible for ensuring that the changes are made to the permanent storage copy of the data. They make these changes to permanent storage either immediately after the update or at a later time

depending on the cache protocol. See *TPF Application Programming* for more information about permanent storage and the relationship to logical record caches.

- *Local cache buffers*, which are buffers that TPF systems allocate in their own storage area. They contain copies of data that is shared among cache users. TPF systems read data from permanent storage to their local cache buffers and write data from their local cache buffers to permanent storage. Each TPF systems that accesses the CF cache structure must have a set of local cache buffers to accommodate the data to be shared. See *TPF Application Programming* for more information about local cache buffers and their relationship to logical record caches.
- *Local cache vector*, which is a vector that provides a way for CF cache users to determine if data in their local cache buffers is valid. There is one local cache vector for each TPF system using the cache. Each vector is divided into separate entries with each entry corresponding to a local cache buffer. Each vector entry contains an indicator that the CF sets to indicate whether the data in the corresponding local cache buffer is valid.

The TPF system supports processor unique caches and processor shared caches:

- *Processor unique cache* contains cache entries that are used by only one processor in a loosely coupled complex.
- *Processor shared cache* contains cache entries that are kept synchronized between all processors in a loosely coupled complex that are using the cache.

Figure 62 shows the elements and their relationship to each other for a processor shared cache. Each piece of shared data can be stored in different locations in the cache system. Copies of shared data are stored in the local cache buffers (fastest access) belonging to each cache user. The shared data also resides on permanent storage (slower access to the data than from the local cache). In general, how quickly you access the data depends on where it is stored.

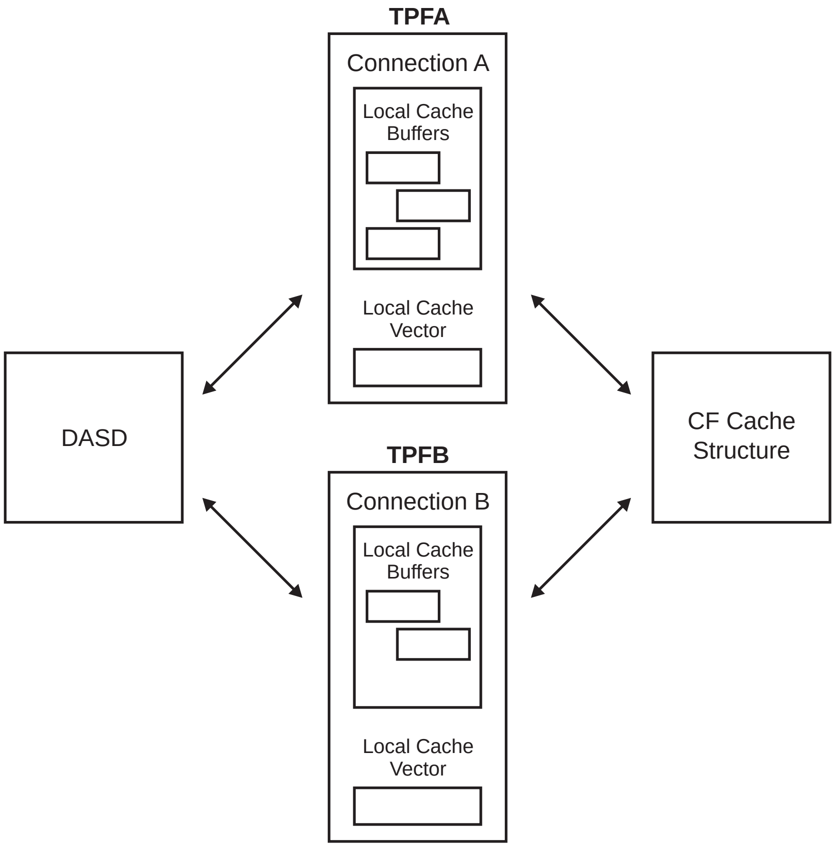


Figure 62. Elements of a Cache System for Processor Shared Cache

Figure 63 shows the elements and their relationship to each other for a processor unique cache.

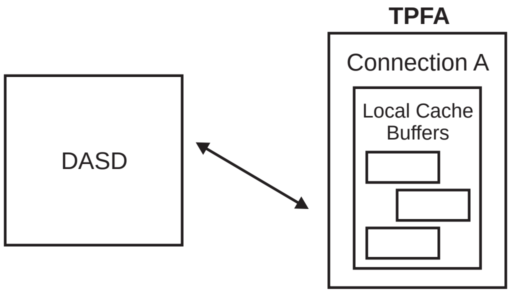


Figure 63. Elements of a Cache System for Processor Unique Cache

Elements of a Coupling Facility Cache Structure

The TPF system supports directory-only CF cache structures. Directory-only cache users ***do not*** store data in the CF cache structure. Rather, directory-only users use the CF cache structure to maintain the consistency of data in their local caches.

The *directory* is a directory for the CF cache structure where the TPF system keeps control information about data shared among cache users. There is one directory entry for each piece of data that is shared. Shared data is maintained in the local cache buffer of each TPF system.

If a directory entry exists in the cache for a piece of shared data (that is, the TPF system has assigned a directory entry to the piece of shared data), the data is said to be *identified* to the CF cache structure. When a piece of shared data is identified to the CF cache structure, each TPF system receives notification through the local cache vector about the validity of the data. As long as the piece of shared data is identified by a directory entry, the CF can be notified that the data associated with the directory entry has been changed and is, therefore, no longer valid. A CF cache structure that contains directory entries but no pieces of shared data is referred to as a *directory-only* cache.

Accessing the Data

A directory-only user needs to access permanent storage frequently.

- Reading a Piece of Shared Data: The TPF system checks the local cache vector entry that corresponds to the piece of shared data to determine if the copy of the data is valid. If the local cache buffer does not contain a valid copy, the TPF system must read from permanent storage.
- Writing a Piece of Shared Data: The TPF system must write to permanent storage and use cross-invalidation to invalidate the local copies of data of other TPF systems.

Maintaining Data Consistency in a Cache System

Each time a connecting TPF system tries to read or write data, the interest of that TPF system is registered in that piece of shared data. It also indicates, in a local cache vector entry the TPF system specifies, that the copy of the piece of shared data is valid. A valid copy of data is one that contains the latest updates to that piece of shared data that other TPF systems might have made.

Registering interest allows the TPF system to ***remember*** that the local cache buffer contains a valid copy of the piece of shared data. If that piece of shared data is changed, the TPF system deregisters interest in that piece of shared data for the others and indicates in their local cache vector entry that the copy is no longer valid. Each connecting TPF system must ensure that the locally cached copy is valid by testing the vector entry associated with that piece of shared data. Each TPF system also needs to ensure that there is serialization of the data between the time the TPF system tests the validity of the piece of shared data and the time when the TPF system makes use of the data.

Registering Interest in a Piece of Shared Data and Validating Local Copies

When interest is registered, the TPF system must specify an entry in the local cache vector that has been assigned to that piece of shared data. The TPF system uses the vector entry to indicate that the associated piece of shared data in the local cache buffer is valid. Figure 64 on page 246 shows a piece of shared data X in the local storage buffer of connecting user A. The piece of shared data is valid because vector entry 2 (the vector entry that connection A assigned to the shared piece of data X) indicates that the data is valid.

The TPF system keeps track of the validity of copies of the shared pieces of data, and the vector entries for each TPF system in the directory entry for each data item in the CF cache structure. In Figure 64 on page 246, the directory for the shared piece of data Z shows that connecting users A and B have registered interest in Z; that is, the connections have valid copies of the shared piece of data Z. If a third connection updates Z in the CF cache structure, the TPF system uses the assigned vector entries (entry 5 for connection A and entry 4 for connection B) to invalidate the local copies belonging to connections A and B.

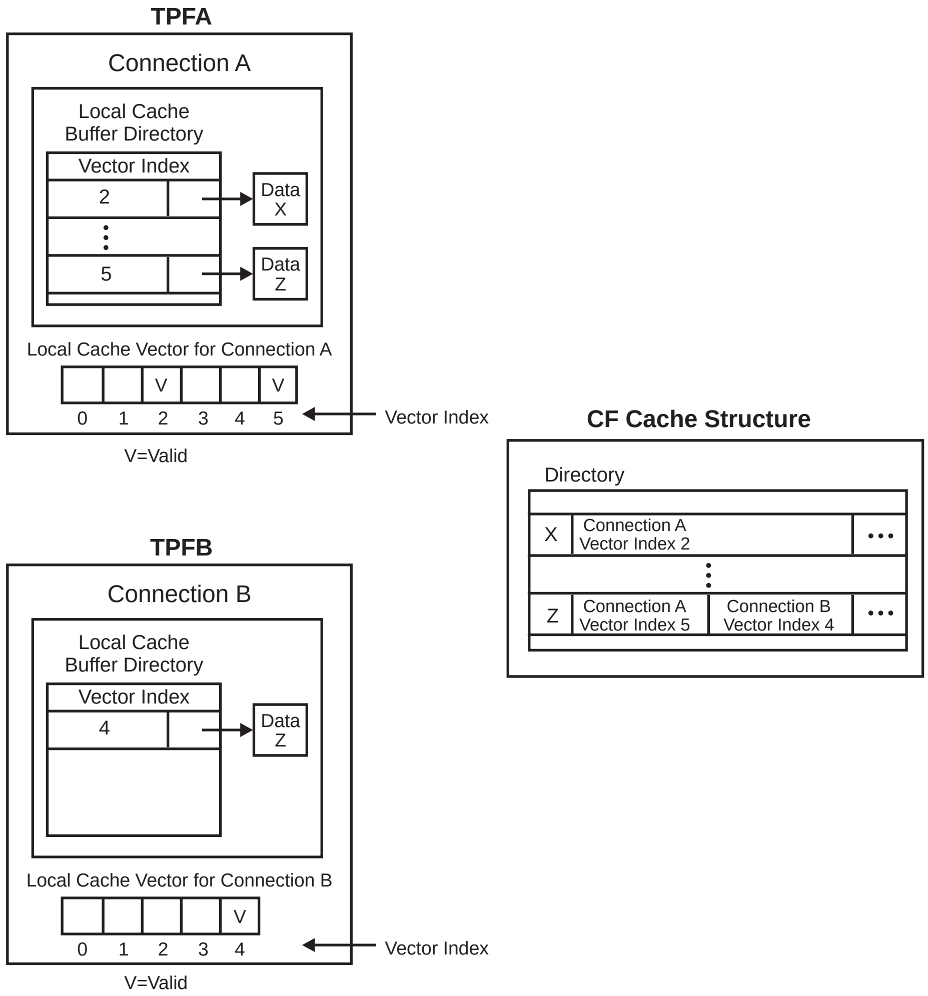


Figure 64. Registered Interest in Shared Pieces of Data

Deregistering Interest in a Shared Piece of Data and Invalidating Local Copies

Figure 65 on page 247 shows what happens when connection A updates the shared piece of data Z. The TPF system invalidates the copy of Z belonging to connection B using local cache vector entry 4 (the vector entry that connection B assigned to Z). Notice also that the CF cache structure directory shows that only connection A has registered interest in Z; connection B has been deregistered.

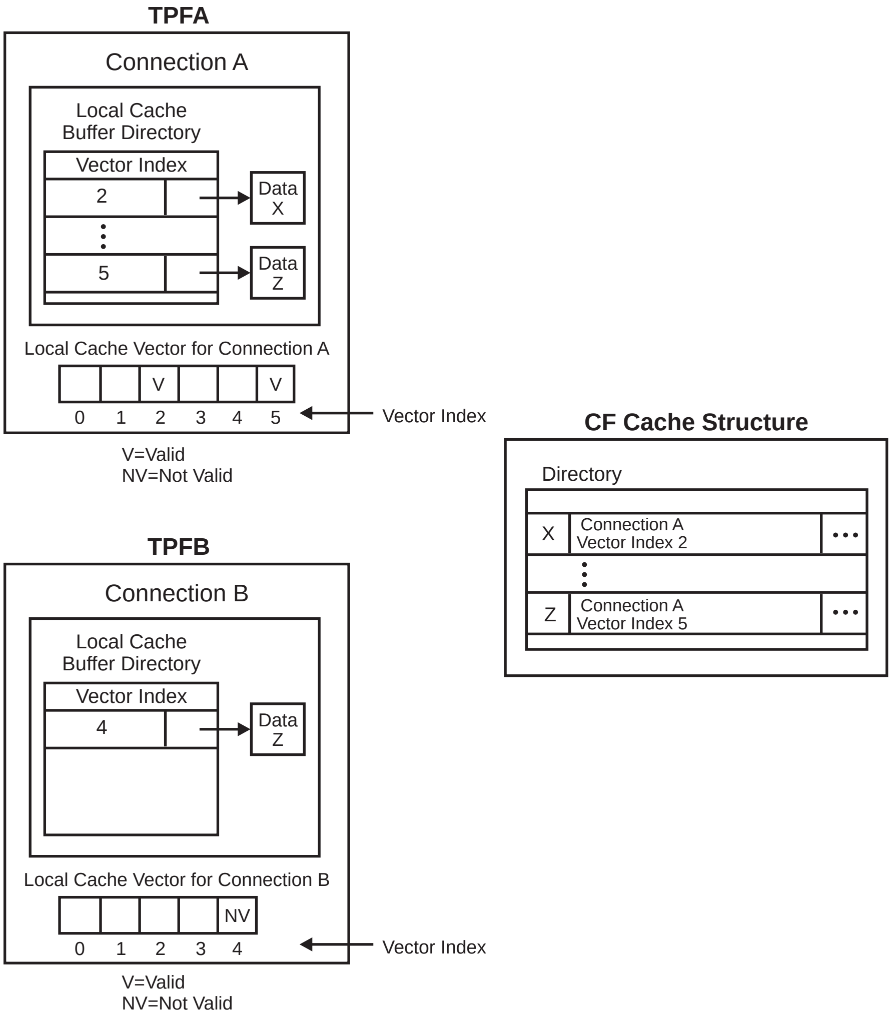


Figure 65. Invalidating a Local Cache Copy of a Shared Piece of Data

Coupling Facility Tables

CF support uses the following tables:

- Coupling facility control table (CFCT)
- Coupling facility status table (CFST)
- Coupling facility trace table (CFTT)
- Message subchannel table (MSCT).

Coupling Facility Control Table

The CFCT is referenced by the CMMCFC CINFC tag and contains information global to CF support such as pointers to other tables and locks for CF processors.

Coupling Facility Status Table

The CFST maintains the status of each CF that was added to a processor configuration. The table contains multiple entries with each entry containing information particular to a CF in the processor configuration. Section 1 of each entry is maintained on file and is rebuilt following each initial program load (IPL).

Coupling Facility Trace Table

The CFTT contains trace data for use by IBM service representatives.

Message Subchannel Table

The MSCT contains information about message subchannels that are detected in the TPF system.

Coupling Facility Blocks

CF support uses the following blocks:

- Coupling facility connection block (CFCB)
- Coupling facility structure block (CFSB)
- Coupling facility request block (CFRB)
- Coupling facility vector block (CFVB).

Coupling Facility Connection Block

The CFCB is a dynamic CF area whose contents are ***not*** maintained on file and are deleted by TPF restart. In addition, your application must reconnect the CFCB following each IPL.

There is one CFCB for each connection of a CF to the TPF system, and that CFCB contains information about the connection. Each time a CF connects to a CF structure, a new CFCB is obtained and added to the corresponding CFCB chains.

Coupling Facility Structure Block

The CFSB is a dynamic CF area maintained in both main storage and fixed file records that contains information about a CF structure. There is one CFSB for each CF structure in any CF of the processor configuration.

Each time a CF structure is created in a CF, a new CFSB is obtained and added to the corresponding CFSB chain. The contents of the CFSB are maintained on file and rebuilt following each IPL

Coupling Facility Request Block

The CFRB is a dynamic CF area that monitors the progress of a CF macro call. Each time a CF macro is issued, a CFRB is obtained to monitor the progress of the macro call. CFRBs are chained as a queue in storage. The contents of the CFRB are ***not*** maintained on file.

Coupling Facility Vector Block

The CFVB is a dynamic CF area whose contents are ***not*** maintained on file. CF support uses CFVBs to track completion of asynchronous CF requests, and to maintain validity of data in local cache buffers.

Storage Dump Format

A CF-related storage dump includes the CFTT.

User-Modified Equates

You can modify the equates that start with ICFU in the ICFEQ segment and the `c$cfeq.h` header file. Use your system configuration and your storage constraints to determine if you want to update the default values before assembling and compiling the CF components in your TPF system.

You can modify the following symbols:

Symbol	**Description**
ICFUMSCTE	The minimum number of MSCT entries that can be allocated. This value is rounded up to fill an integral number of 4-KB pages before allocating the MSCT. This value must be nonzero. The default is 250.
ICFUCFTTE	The minimum number of CFTT entries that can be allocated. This value is rounded up to fill an integral number of 4-KB pages before allocating the CFTT. This value must be nonzero. The default is 1000.
ICFUMAXTIME	The number of seconds that may elapse between warnings when the CF lock cannot be obtained.

Coupling Facility Locking Functions

This section will help you to understand how to use the serialized list structure as well as provide information about the CF locking functions and the format of the CF lock.

Overview

As discussed previously in “Coupling Facility List Structure” on page 235, a serialized list structure is a list structure that contains a lock table. The lock table is an array of exclusive locks whose purpose and scope are defined by the application. Lock table locks can provide a serialization mechanism for lists, list entries, or any other list structure entity you designate. The first connector to a CF list structure specifies whether it is to be a serialized list structure and, if so, the number of lock entries to be allocated in the lock table. Figure 60 on page 236 shows a serialized list structure.

Coupling Facility Lock Format

The CF lock is used to serialize operations on the CF and is 16 bytes long. Figure 66 shows the format of the CF lock.

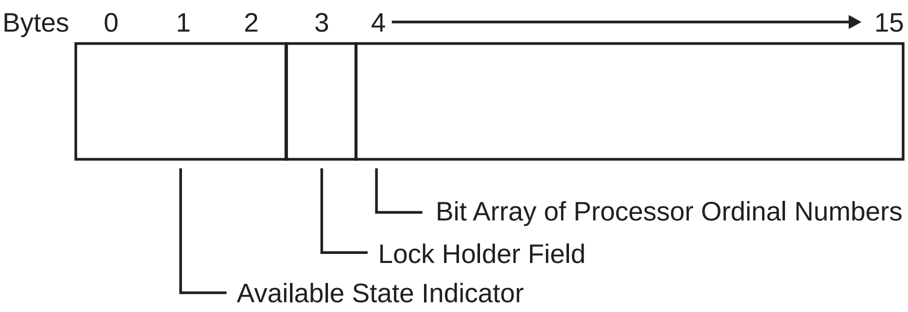

Bytes
0
1
2
3
4
15
Bit Array of Processor Ordinal Numbers
Lock Holder Field
Available State Indicator

Figure 66. Format of the CF Lock

In the CF lock:

- Bytes 0 to 2 are known as the *available state indicator* and can contain either the characters ’TPF’ or X'000000'. For example, bytes 0 to 2 will contain ’TPF’ when any bit in the bit array of processor ordinal numbers is nonzero. When resetting a bit in the bit array results in all bits being zero, the available state indicator changes from ’TPF’ to X'000000'.

  The CF lock is considered corrupted when the available state indicator does ***not*** contain the characters ’TPF’ and the CF lock is ***not*** set to all zeros.
- Byte 3 is known as the *lock holder field*. The *lock holder field* indicates one of the following to the user:
  – The CF lock is ***not*** being held by a processor and can be obtained for use. When that occurs, the lock holder field is set to X'00'.
  – The CF lock is currently being held by a processor. When this occurs, the lock holder field is set to the processor ordinal number of the processor holding the CF lock plus 1. For example, if processor B has a processor ordinal number of 0, the lock holder field would be set to X'01' (0 + 1).
- Bytes 4 to 15 are a bit array of processor ordinal numbers. When you add a CF to a processor configuration using the ZMCFT ADD command, the processor ordinal number for that processor is set on in bytes 4 to 15 of the CF lock. The processor ordinal number determines which bit gets turned on in the bit array.

The following scenarios illustrate how the content of the CF lock changes when different CF operations are performed or various system states occur.

- When you add a CF to a processor configuration using the ZMCFT ADD command, the CF is put in an available state. An *available state* is the state a CF is in when all CF commands are processed normally. When a CF is put in available state, the available state indicator contains the characters ’TPF’. Figure 67 shows the format of the CF lock when a CF is in available state.

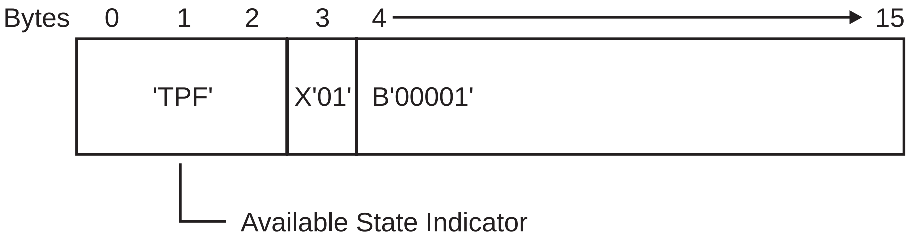


Figure 67. Format of the CF Lock When a CF Is in Available State

- When bytes 0 to 15 are set to all zeros, the CF is in a nonavailable state. A *nonavailable state* is the state a CF is in when only certain CF commands are processed normally; all other CF commands are suppressed. Figure 68 shows the format of the CF lock when a CF is in a nonavailable state.

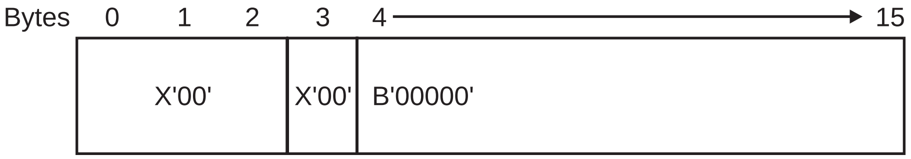


Figure 68. Format of the CF Lock When a CF Is in a Nonavailable State

- When the lock holder field of the CF lock is set to X'00', no processor is currently holding the CF lock and you can obtain it now. Figure 69 shows the format of the CF lock when the lock holder field is set to X'00'.

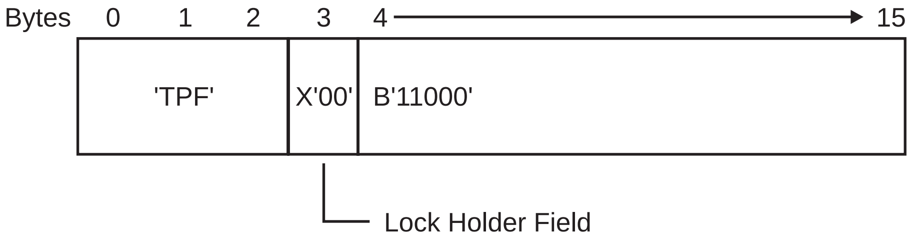


Figure 69. Format of the CF Lock When the Lock Holder Field Is Set to Zero

When the lock holder field is not set to X'00', it is set to*P*+1, where *P* is the processor ordinal number of the processor holding the CF lock. For example, if processor C has a processor ordinal number of 1, the lock holder field would be set to X'02' (1 + 1). Figure 70 shows the format of the CF lock when the lock holder field is set to the processor ordinal number of processor C.

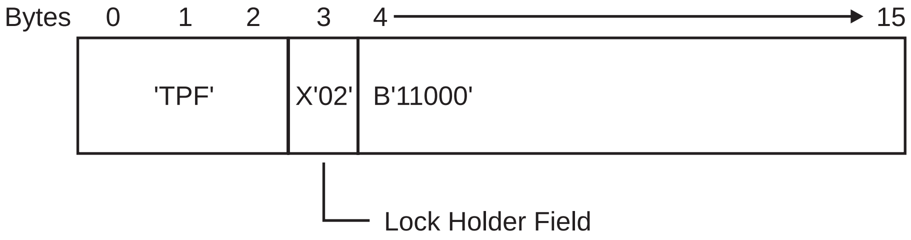


Figure 70. Format of the CF Lock When the Lock Holder Field Is Set to the Processor Ordinal Number for Processor C

Defining Exit Routines

CF support provides the list transition exit routine, which plays a critical role in the operation of the CF list structure. Users provide the address of the exit routine when they issue the CFCONC macro to connect to the CF list structure.

The *list transition exit routine* informs users when lists they are monitoring change from an empty state to a nonempty state. The affected lists are not identified; this exit routine must issue the CFVCTC macro to determine which lists changed from an empty state to a nonempty state.

Conditions at Entry

- IBM recommends that you place the list transition exit routine in the control program (CP).
- Connection services passes information to the list transition exit routine in the list transition exit parameter list (LEPL). The registers at entry to this exit routine are:

R1	Address of the LEPL.
R14	Address to which control is to be returned.
R15	Address of the entry point for the list transition exit routine.

In addition to these registers, standard CLNKC linkage must be followed.

System Conditions on Entry

System state	Supervisor
Protect key	0
Address space	System virtual memory (SVM)
Interrupt status	Disabled for external and input/output (I/O) interrupts.

Programming Considerations at Entry

1. Provide the address of the list transition exit routine using the LISTTRANEXIT parameter when you issue the CFCONC macro to connect to the CF list structure. The exit routine may receive control before control is returned to you from the CFCONC macro. Therefore, ensure that you have the exit routine established along with any control structures that are necessary for completing the processing of the exit routine before issuing the CFCONC macro.

   When the exit routine receives control, it receives information about the request and the outcome in the LEPL. The LEPL is mapped by the ICFLEPL DSECT, which is generated by the ICFPL macro with the LEPL=YES parameter coded.
2. If you use the list transition exit routine to monitor several lists, be aware that the exit routine is given control whenever any list that you monitor in this way changes from empty state to nonempty state. To determine which monitored list changed its state, issue the CFVTC macro with either the REQUEST=TESTLISTSTATE or REQUEST=LTVECENTRIES parameter coded to check the vector entry for each monitored list.
3. The time interval involved when detecting and responding to a list transition exit routine introduces several timing considerations, particularly if multiple connections are monitoring the same list:
   - If multiple connections are monitoring the same list, the first connection to respond to the exit routine could empty the list before other connections test the list notification vector or check the list. Depending on when the other connection emptied the list, either of the following could occur:
     - The exit routine could receive control, test the list notification vector, and find no nonempty lists.
     - The exit routine could receive control, test the list notification vector, find that the list is in a nonempty state, and then try to read a list entry from the list and find it empty.
   - There is a possible delay between the time a list changes from empty state to nonempty state and the time its list notification vector is updated. Changes to the list notification vector are made in the order in which the corresponding list transitions occur. However, timing of the updates is not guaranteed.
   - There is a possible delay between the time a list transition occurs and the time the exit routine receives control.
   - A list transition exit may receive control even though the monitored list has not changed.
4. You can only access the LEPL data area while the list transition exit is running. If you want to save the LEPL information for later processing, make a copy of it before the exit returns.

Conditions on Return

The list transition exit routine must return control to the address that is contained in register 14 (R14) on entry. The contents of all the registers must be restored before returning control to the address in R14. The CEPL and the LEPL includes a 16-fullword save area in which you may choose to save the registers of the caller. In addition, you cannot change the program status word (PSW) or the I/O or external interrupt masks.

Programming Considerations on Return

The list transition exit routine must return to the caller without giving up control.

Coupling Facility Record Lock Support

The limited lock facility (LLF) and the concurrency filter lock facility (CFLF), which are two external lock facilities (XLFs) supported by the TPF system, were required to control access to data shared by two or more processors in a loosely coupled complex. CF record lock support provides the option of using one or more CFs as XLFs.

CF record lock support offers significant flexibility in using CFs as XLFs in your CF locking configuration. Your CF locking configuration may be dynamically modified by adding or deleting CFs. When a CF is added to or deleted from the CF locking configuration, the TPF system automatically redistributes any CF locks to balance the locking workload across all available CFs. You can add as many as 32 CFs to your CF locking configuration for a high degree of availability. The CFs in your CF locking configuration can be used in addition to or instead of LLF and CFLF. In addition, the CFs in your CF locking configuration can simultaneously be used for nonlocking workloads. Using CFs in a locking configuration can eliminate the need for LLF or CFLF XLFs, giving you greater flexibility when selecting and implementing new module control units (CUs).

All online modules in a loosely coupled complex must use an XLF for locking to control access to shared data. You may specify which online modules will use CFs for locking even if those modules are connected to a CU with an LLF or CFLF. Modules can be migrated to use CFs for locking either individually, in groups, or by migrating all online modules at once. Note that the lock residency of any duplicate module is configured automatically to be identical to that of the corresponding prime module.

See *TPF Migration Guide: Program Update Tapes* for more information about CF record lock support and see*TPF Concepts and Structures* for more information about CF record lock support, the LLF, CFLF, and XLFs.

Concepts for Coupling Facility Record Lock Support

Before continuing with the information in this chapter, review the conceptual information and terminology provided in “Data Sharing Concepts and Terminology” on page 223 and “Coupling Facility List Structure Concepts” on page 235.

Coupling Facility List Structures for Coupling Facility Locking

CF record lock support creates the following CF list structures for CF locking on each CF in the locking configuration:

- ITPFLK1_*xxxxxx*
- ITPFLK2_*xxxxxx*

Where *xxxxxx* is the complex name.

Any event that changes the status or configuration of the TPF system and impacts the distribution or location of CF locks on the external locking facility (XLF) can require that CF locks are reassigned from one XLF to another. Events that can trigger lock recovery include:

- Changing the status of a module from offline to online or online to offline
- Completing an all file-copy operation
- Adding or removing a CF from the CF locking configuration

- Changing an assignment of lock residency between a module and a CF
- Failure of a CF that was actively participating in CF locking.

Coupling Facility Record Lock SupportCommands

The commands are used to display entries in the coupling facility locking table (CFLT) and manage the CFs in a locking configuration. The following lists the CF locking commands and their functions:

Function	Description
ZCFLK ADD	Adds a CF to the CF locking configuration, which then enables the CF for use as an XLF so new locks can be stored on it.
ZCFLK DELETE	Removes a CF from the CF locking configuration. All CF locks stored on this CF are redistributed automatically among the remaining CFs in the CF locking configuration on a module-by-module basis. As a result, CF locks can no longer be stored on this CF.
ZCFLK DISPLAY	Displays information about the CF locking configuration.
ZCFLK INITIALIZE	Initializes the CF locking configuration.
ZCFLK MIGRATE	Changes the lock residency of a module from a CFLF locking control unit (CU) to a CF, or from a CF to a CFLF locking CU.
ZDLCK DISPLAY	Displays locks in a CF.

See *TPF Operations* for more information about these commands.

Using the Coupling Facility Record Lock Support Commands

This section shows a scenario of how you might use the CF commands to manage locks in a CF locking configuration by providing examples that span a series of commands.

Example 1: In the following example, two CFs are added to a processor configuration. First, a CF named CFONE, which is attached to symbolic device address (SDA) FF00, is added. Then, a CF named CFTWO, which is attached to SDA FE00, is added. Both CFs are then added to the locking configuration. The size of the CF list structure for locking on both CFs is 1200 4-K blocks. All the online modules on both the basic subsystem (BSS) and the WP subsystem are then migrated to use the CFs for locking. The general data set (GDS) modules on both the BSS and the WP subsystem are then migrated to use the CFs for locking.

```
User:    ZMCFT ADD CFONE FF00
System:  MCFT0001I 18.54.45 CFMADD - COUPLING FACILITY CFONE ADDED -
                            2 PATHS EXIST

User:    ZMCFT ADD CFTWO FE00
System:  MCFT0001I 18.54.45 CFMADD - COUPLING FACILITY CFTWO ADDED -
                            2 PATHS EXIST

User:    ZCFLK ADD CFONE SIZE 1200
System:  CFLK0002I 18.54.45 CFLO - COUPLING FACILITY CFONE WAS ADDED TO THE
                            LOCKING CONFIGURATION

User:    ZCFLK ADD CFTWO SIZE 1200
System:  CFLK0002I 18.54.45 CFLO - COUPLING FACILITY CFTWO WAS ADDED TO THE
                            LOCKING CONFIGURATION

User:    ZCFLK MIGRATE ONL TO CF
System:  CFLK0004I 18.54.45 CFLW - ZCFLK MIGRATE PROCESSING COMPLETE

User:    WP/ZCFLK MIGRATE ONL TO CF
System:  CFLK0004I 18.54.45 CFLW - ZCFLK MIGRATE PROCESSING COMPLETE

User:    ZCFLK MIGRATE GDS TO CF
System:  CFLK0004I 18.54.45 CFLW - ZCFLK MIGRATE PROCESSING COMPLETE

User:    WP/ZCFLK MIGRATE GDS TO CF
System:  CFLK0004I 18.54.45 CFLW - ZCFLK MIGRATE PROCESSING COMPLETE
```

Example 2: In the following example, a range of modules is migrated to use CF for locking.

```
User:    WP/ZCFLK MIGRATE 01E.021 TO CF
System:  CFLK0004I 20.54.45 CFLW - ZCFLK MIGRATE PROCESSING COMPLETE
```

Example 3: In the following example, one module is migrated to use a control unit (CU) for locking.

```
User:    ZCFLK MIGRATE 047 TO CU
System:  CFLK0004I 18.54.45 CFLW - ZCFLK MIGRATE PROCESSING COMPLETE
```

Example 4: In the following example, the CF named CFTWO is deleted from the locking configuration. By mistake, an attempt is made to delete the CF named CFONE from the locking configuration before all modules have been migrated to use CUs for locking.

```
User:    ZCFLK DEL CFTWO
System:  CFLK0003I 18.54.45 CFLDEL - COUPLING FACILITY CFTWO WAS DELETED FROM
                            THE LOCKING CONFIGURATION

User:    ZCFLK DEL CFONE
System:  CFLK0031E 18.54.45 CFLDEL - DELETE ENDED - COUPLING FACILITY CFONE
                            CANNOT BE DELETED - IT IS THE LAST CF IN THE
                            LOCKING CONFIGURATION AND LOCKS STILL RESIDE ON IT
```

Coupling Facility Locking Table

CF record lock support uses the coupling facility locking table (CFLT). The main storage CFLT is referenced by the CMMCFL CINFC tag and contains information essential to maintain correct locking on a CF in the CF locking configuration. The CFLT on file is maintained in the #CF2LR fixed file record.

User Modification Considerations

This section provides information to help you determine the CF list structure size for CF locking and considerations for when you change the lock name.

Determining the Coupling Facility List Structure Size for Locking

The size of the CF list structure you will use for locking is specific to your CF locking configuration. See your IBM service representative for guidance when determining the size of the CF list structure for locking.

Do the following if you need to change the size of the CF list structures you are using for locking:

1. Enter the ZCFLK DELETE command to remove the CF from the CF locking configuration. The lock residency for all CF locks stored on this CF is recalculated automatically among the remaining CFs in the CF locking configuration on a module-by-module basis. As a result, CF locks can no longer be stored on this CF.
2. Enter the ZCFLK ADD command with a new value specified for the SIZE parameter to add the CF to the CF locking configuration, which then enables the CF for use as an XLF so new locks can be stored on it.

See *TPF Operations* for more information about these commands.

Changing the Lock Name

If you change the lock name, you must provide a routine in the CFL1 copy member at CFL1VALK to prevent errors from occurring.

Changing the Name of Your Complex

If you change the name of your complex and then perform an initial program load (IPL) of the TPF system, you must ensure that all CF list structures for locking have been deleted and added again.

Index

File Number: S370/30XX-30
Program Number: 5748-T14

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.