zVSAM V2 Design and Logic Manual V2.3

Table of Contents

Items marked ### are not yet complete

Introduction

This document describes the structure of the zVSAM component of the z390 assembler and emulator. It consists of the following parts:

Copyright Notice

This document Copyright 2018 – z390 development team

z390 is free software; its associated documentation is equally free. You can redistribute and/or modify both software and documentation under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version

z390 is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with z390; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Acknowledgements

z390's purpose is to provide a source-level compatible assembler, linker and run-time emulation engine for IBM's High-Level Assembler environment. By extension, the zVSAM component described in this document also aims at providing source-level compatibility. As a consequence, all source-level interfaces necessarily mimic the IBM-provided (and IBM-copyrighted) interfaces as described by IBM in publicly accessible documents

The logic and implementation behind these interfaces, however, was developed independently from IBM and is the product of the joint efforts of our team of volunteer developers

Source-level compatibility is a primary goal not only for z390 and zVSAM, but also for other z390 components such as zCOBOL and zCICS.

All IBM publications and software we refer to in this document are copyright IBM Corporation with no exception

The drawings in this document have been made using the draw.io software
As part of the open source for z390 the xml and jpg documents describing these drawings are available with every distribution of z390 that contains this document

Terminology

The reader is assumed to have at least some familiarity with IBM VSAM, to the extent that most of the following acronyms and terms are understood:

-ACB	Access Control Block
-AIX	Alternate IndeX
-CBMR	Control Block Modification Request (zVSAM)
-CI	Control Interval
-ELIX	Extended Level IndeX – extra index level for non-unique AIX (zVSAM)
-ESDS	Entry Sequenced Data Set
-IBM	International Business Machines Corp., USA
-KSDS	Key Sequenced Data Set
-Path	Access to a base cluster, usually through an AIX
-RBA	Relative Byte Address
-RDW	Record Descriptor Word in IBM-defined format
-RLF	Record Length Field – (zVSAM)
-RPL	Request Parameter List
-RRDS	Relative Record Data Set
-RRN	Relative Record Number
-SPX	Segment Prefix (zVSAM)
-VSAM	Virtual Storage Access Method
-XRBA	Extended Relative Byte Address
-XLRA	Extended Logical Record Address (zVSAM)
-zACB	zVSAM equivalent of the ACB
-zEXLST	zVSAM equivalent of the EXLST
-zRPL	zVSAM equivalent of the RPL

In this document we also use the following terms. The ones that are used by IBM as well, are intended to have the same meaning they do in IBM manuals

Area a section of storage with a defined layout, depending on the type of Area
Block zVSAM equivalent of a Control Interval
Cluster a set of files that logically belong together
Component either a data component or an index component of a cluster
Element a primary key or XRBA in an AIX record
File a single file as seen by the hosting operating system
Foxes a value consisting of all high-values i.e. a value of all X'FF' bytes.
List a structure holding items that are linked together by pointers
Segment a portion of a record in a spanned dataset
Segmented a record that has been split into segments in a spanned dataset
Spanned an attribute of a dataset that allows records to be split into segments
Sphere a cluster and all associated AIXs
Table a structure holding items that are physically adjacent

Compatibility

As this document relates to zVSAM V2, there are two type of compatibility we need to consider. On the one hand we have designed zVSAM to be compatible with IBM VSAM. And on the other hand we need to consider compatibility with z390's zVSAM V1 – the prior implementation of zVSAM in the z390 environment

zVSAM compatibility with IBM VSAM

Our z390 implementation of zVSAM V2 is intended to be source-level compatible with IBM VSAM. This has the following consequences:

1. IBM VSAM documentation with regards to macros and interfaces applies to zVSAM with the exception of parameters and options not supported by zVSAM. Where zVSAM differs in behaviour this is noted in this document. Please refer to the macro descriptions for details
2. Control Blocks (such as ACB, RPL and some others) are not compatible. zVSAM has its own structures. A side effect of this may be that a program's assembled object code may be different in size than on your IBM operating system. On rare occasions you may need an additional base register when porting your program either way
3. As a rule of thumb, a program using VSAM can be ported to z390 and should be able to assemble, link and run without modification – provided it uses only the VSAM features and options that zVSAM supports. And provided the program does not run out of addressability due to different control block lengths

zVSAM V2 compatibility with zVSAM V1

The user of z390's zVSAM component should be aware that zVSAM V2 as described in this document is not compatible with the pre-existing zVSAM V1. We – the development team – apologize for the inconvenience this may cause

We have taken the following measures to facilitate the transition from zVSAM V1 to zVSAM V2:

1) We have introduced a new z390 option: ZVSAM which indicates which version of zVSAM you want z390 to use. It takes the following forms:
ZVSAM(0) – zVSAM usage is disallowed
ZVSAM(1) – zVSAM V1 is enabled, zVSAM V2 is disabled
ZVSAM(2) – zVSAM V2 is enabled, zVSAM V1 is disabled
For maximum compatibility the default is set to ZVSAM(1).
The default will be changed to ZVSAM(2) in a future release of z390

2) To convert your zVSAM V1 clusters to zVSAM V2 you'll have to take the following steps:
- unload the existing data from their clusters using REPRO
- reload your data from your unload files, using ZREPRO
For details on how to use REPRO, please refer to the "z390_VSAM_User_Guide"
For details on how to use zREPRO, please refer to the "z390_zVSAM_zREPRO_User_Guide"

3) For zVSAM V1 and zVSAM V2 there are distinct macro libraries, MACVSAM1 and MACVSAM2
To use the correct zVSAM maclib, specify the correct version in your maclib concatenation

4) If a program is to run with ZVSAM(2), then all submodules that contain an OPEN macro must be re-assembled using MACVSAM2, even those that only use QSAM

API for Assembler and zCOBOL programs

ACB-based interfaces

The ACB is the primary interface for operations at the cluster level.
Each cluster is represented by an ACB

The ACB interface consists of an ACB control block, possibly an Exit list Control Block, and a set of macros to manage and manipulate the ACB and EXLST control blocks. These macros can be used in your assembler programs. For zCOBOL and/or other higher-level languages, these macros will be generated from specifications for the files as appropriate in the host language's syntax

The following macros are provided for assembler programs:

- ACB
- ACBD
- CBMR
- GENCB BLK=ACB
- MODCB ACB=
- SHOWCB ACB=
- TESTCB ACB=

Note: The ACB macro defines a statically allocated ACB. This macro is primarily intended for use in non-re-entrant programs. GENCB BLK=ACB should be used to create an ACB in dynamically acquired storage, or in private static storage. MODCB ACB= can be used to modify an existing ACB, whereas SHOWCB ACB= can be used to query specific fields of an ACB and TESTCB ACB= can be used to validate specific fields of an ACB

- EXLST
- EXLSTD
- GENCB BLK=EXLST
- MODCB EXLST=
- SHOWCB EXLST=
- TESTCB EXLST=

Note: The EXLST macro defines a statically allocated EXLST. This macro is primarily intended for use in non-re-entrant programs. GENCB BLK=EXLST should be used to create an EXLST in dynamically acquired storage, or in private static storage. MODCB EXLST= can be used to modify an existing EXLST, whereas SHOWCB EXLST= can be used to query specific fields of an EXLST and TESTCB EXLST= can be used to validate specific fields of an EXLST

- OPEN
- CLOSE

Note: OPEN and CLOSE macros can be used to open and close either sequential files represented by a DCB and/or zVSAM files represented by an ACB

A description of these interfaces as implemented for z390 and zVSAM is detailed in the next chapters

ACB Macro

The ACB macro will generate an ACB and initialize it according to the parameters specified on the macro invocation

Direct access to subfields in the ACB is discouraged. Use SHOWCB ACB=, TESTCB ACB= and/or MODCB ACB= to inspect, test, and/or modify the ACB's content

All keywords on the ACB macro are optional. Before the cluster is opened, all ACB values can be modified using MODCB ACB=, or by changing the ACB directly. The latter is not recommended, as it is not guaranteed to be portable or compatible with future versions of zVSAM

The table below shows how the ACB macro can be coded

Opcode	Operand	Remarks
[label] ACB	[AM=VSAM]	Designates this ACB as a zVSAM ACB
	[DDNAME=ddname]	DDNAME: name of an environment variable in the host OS holding the name of the cluster to be processed See notes here
	[PASSWD=ptr]	Pointer to password for the cluster Points to a single byte length followed by the password eg. X'05',C'ABCDE'
	[EXLST=ptr]	Pointer to an exit list. Please see the EXLST macro description for details here
	[MACRF=(keywd_list)]	List of keywords specifying processing options See table below for valid keywords here
	[BUFSP=nr]	Max amount of storage (in bytes) to use for buffers See notes here
	[BUFND=nr]	Number of data buffers to allocate for this ACB Specify a number between 1 and 65535
	[BUFNI=nr]	Number of index buffers to allocate for this ACB Specify a number between 1 and 65535
	[RMODE31=keyword]	Indicates whether buffers and/or control blocks can be allocated above the line See notes here
	[STRNO=1]	Number of concurrent requests allowable for this ACB Specify a number between 1 and 255
	[BSTRNO=nr]	Beginning number of concurrent requests allocated to this ACB when a path is opened. Only applies if MACRF=NSR Specify a number between 0 and 255
	[MAREA=ptr]	Not supported – future option Keyword is flagged as ignored with a warning message
	[MLEN=nr]	Not supported – future option Keyword is flagged as ignored with a warning message
	[RLSREAD=keyword]	Not supported – future option Keyword is flagged as ignored with a warning message
	[SHRPOOL=nr]	LSR shared pool number – future option

ACB MACRF keywords

Keyword subset	Keyword	Remarks
[ADR KEY]	ADR	Addressed access to ESDS by (X)RBA Using (X)RBA to access a KSDS is not supported
	KEY	Keyed access to a KSDS RRN access to an RRDS
	CNV	Not supported. Keyword is flagged with a warning message
[DFR | NDF]	DFR	Allow writes to be deferred
	NDF	Do not defer writes
[DIR SEQ SKP	DIR	Direct access to ESDS, KSDS or RRDS
	SEQ	Sequential access to ESDS, KSDS or RRDS
	SKP	Skip sequential access to KSDS or RRDS Only for keyed access. Allows the use of POINT
[IN OUT]	IN	Read only access for ESDS, KSDS or RRDS
	OUT	Both read and write access for ESDS, KSDS or RRDS
[NIS | SIS]	NIS	Normal Insert Strategy for KSDS
	SIS	Sequential Insert Strategy for KSDS
[NRM | AIX]	NRM	DDNAME indicates cluster to be processed
	AIX	DDNAME of a path to access an AIX directly, rather than using it to access records in the underlying base cluster
[NRS | RST]		Not supported. Keyword is flagged with a warning message
[LSR | GSR | NSR | RLS]		Local, Global or no Shared Buffers. RLS is not supported
[NUB/UBF]		Not supported. Keyword is flagged with a warning message
[CFX/NFX]		Not supported. Keyword is flagged with a warning message
[DDN/DSN]		Not supported. Keyword is flagged with a warning message
[ICI/NCI]		Not supported. Keyword is flagged with a warning message
[LEW/NLW]		Not supported. Keyword is flagged with a warning message

With the exception of the DDNAME parameter explained below, all supported parameters are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual

DDNAME= notes
DDNAME is required before open is executed. If DDNAME is not supplied on the ACB macro, the label used on the ACB macro is used as DDNAME. If neither is specified, a proper value must be supplied by using MODCB ACB=

In zVSAM the DDNAME refers to the name of an environment variable in the host OS. This variable in turn should contain the path and qualified filename of the cluster to be opened. The qualifier is the name of an environment variable in the host OS and is the path to the assembled catalog
For more information on zVSAM catalogs, please refer to the "z390_zVSAM_Catalog_User_Guide"
For more information on environment variables, please refer to the “z390_zVSAM_zREPRO_User_Guide”

BUFSP= notes

Maximum buffer space in virtual storage for this cluster

This is the combined size in bytes of all buffers allocated for this cluster. If (BUFND + BUFNI) * Block_size exceeds the value specified for BUFSP, then BUFND and BUFNI will be reduced proportionally to keep the total allocation below the limit specified in the BUFSP parameter

RMODE31= notes

Specifies whether buffers and/or control blocks should be allocated below or above the 16M line:

NONE	Control Blocks and buffers below 16M
CB	Control Blocks above or below 16M, buffers below 16M
BUFF	Control Blocks below 16M, buffers above or below 16M
ALL	Control Blocks and buffers above 16M or below 16M

The default for RMODE31 is NONE

OPEN macro

A cluster needs to be opened before it can be processed. The open macro is used to open one or more clusters and/or one or more sequential files in a single call

Opcode	Operand	Remarks
[label] OPEN	(entry[,entry]...)	Each cluster or file requires an entry of two parameters
	[MODE=24/31]	Residency mode of all control blocks involved. Specify 31 if any resides above the line
Entry format:	addr,(options)	Address of ACB or DCB, followed by a list of options (for DCB only). For ACB omit the list of options.
	[MF=I or omitted]	Use standard form of OPEN
	[MF=L]	Use list form of OPEN
	[MF=(E,addr)]	Use execute form of OPEN

All supported parameters are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual

OPEN macro parameters

entry The OPEN macro accepts a list of entries. Each entry consists of two consecutive parameters: an address and an optional list of options

address The address can be specified as an A-type address or as a register. If a register is coded the register number or name must be enclosed in parentheses. The address can be either the address of a DCB or the address of an ACB

options For a DCB options may be encoded according to the z390_File_Access_Method_Guide
For an ACB the options list is ignored and should be coded as an omitted parameter
Any options (e.g. IN/OUT) are taken from the ACB, not the open parmlist

MF=I or omitted
An open parmlist is generated inline, plus a call to the OPEN SVC using the parmlist

MF=L An open parmlist is generated inline

MF=(E,addr) Code to modify/populate the open parameter list at the indicated address, which may be a relocatable constant or a (register), plus a call to the OPEN SVC using the parmlist

OPEN logic

Open logic has two major components: the open macro and the actual run-time logic to execute a request to open a file or a number of files

Open parameter list entries have two different formats depending on the MODE parameter.
When MODE=24 then each entry is one fullword
When MODE=31 then each entry is two fullwords

Only one SVC 19 is generated for each OPEN macro (MF=I or E)
The list format and input to OPEN depend on MODE=

```
MODE=24  AL1(option),AL3(DCB/ACB address)
         R1 points to the list

MODE=31  AL1(option),XL3'00',AL4(DCB/ACB address)
         R0 points to the list and R1=0

option=X'40'   INPUT
option=X'20'   OUTPUT
option=X'60'   UPDATE
The last entry has the X'80' bit on in option
The option is ignored when opening an ACB
```

OPEN execution logic

OPEN execution logic is implemented as a Java routine
This logic consists of the following elements:

Action	Details
Determine type of parameter list	31-bit entries, addressed by R0, if R1 = 0 24-bit entries, addressed by R1, if R1 <> 0
loop over all entries in the parameter list	End-of-list is indicated in the option byte of the entry
- check pointer: ACB or DCB	First byte = X'A0' => ACB V1 First four bytes = C'zACB' => ACB V2 First four bytes = C'DCBV' => DCB Otherwise => Error
- if DCB invoke DCB open routine	OPEN logic for DCB is beyond the scope of this document
- if ACB validate ACB	ACBID <> X'A0' => Error ACBSTYP <> X'10' => Error ACBVER <> X'02' => Error ACB V1/V2 <> ZVSAM(n) parm => Error
- if ACB valid invoke VSAM open routine	
- next entry or end-of-loop	If bit 0 of an entry is on, terminate loop

OPEN logic for ACB handles a single ACB and proceeds as follows:

Action	Details
Check ACB status	If ACB already open, issue error and fail open
Copy ACB to newly created FCB	FCB is the java-equivalent of the ACB
Extract DDNAME	Copy ACBDDNM field from ACB/FCB
Find actual file name	Retrieve host variable with name matching ACBDDNM If not available: issue error and fail open
Validate against catalog	Find the file name in the catalog. If missing: issue error See note here
Issue OS open against file	Read-only if ACB specifies MACRF=IN Update/extend otherwise If unsuccessful issue error and fail open
Allocate buffer for prefix block	Save buffer address in FCB
Read first 4096 bytes into buffer	If read fails, issue error
Validate block header and footer	If BHDREYE <> C'HDR' issue error If BFTREYE <> C'FTR' issue error If BHDRSEQ# <> BFTRSEQ# issue error If BHDRVER <> X'02' issue error If BHDRSELF <> foxes issue error If BHDRPREV <> foxes issue error If BHDRNEXT <> foxes issue error If BHDRFLGS <> X'80' issue error

Action	Details

Validate prefix area	if PFXEYE <> C'zPFX' issue error
	if filename <> PFXDNAM issue error
	if file's path <> PFXDPAT issue error
	if PFX_INDX is on issue error

Validate counters area	if CTREYE <> C'zCTR' issue error
Validate prefix against catalog	Only if no errors detected thus far: compare cluster type compare lrecl compare blocksize compare key offset compare key length
Fail open on error	If any error was detected: - request OS to close the file - free the prefix buffer - set buffer pointer in FCB to zeros - fail the open request
Issue OS open against index file	If PFXXNAM@ is non-zero then open the indicated index file; read-only if ACB MACRF=IN for input/update/extend otherwise Read index header block and repeat all validations with the following modifications: - if PFX_INDX is off rather than on issue an error
Fail open on error	If any error was detected: - request OS to close the files - free the prefix buffers - set buffer pointer in FCB to zeroes - fail the open request
Create data buffers	Based on ACBBUFND
Create index buffers	At least ACBBUFNI in total Exactly one for the root block At least 4 for each other index level
Open component	What is opened depends on what type of component the ACB points to A path may imply opening of the base cluster and/or AIXs Repeat the open process for each component File names and other info to be gathered from the catalog The table on the next page has the permutations of component types

Note: The environment variables take the following form

```
SET ddname=drive:\path\catalog.filename
SET catalog=drive:\path
```

The ddname variable may only contain one dot

Implied OPEN table

This table has the permutations of component types, indented entries are implied processing

Open component	MACRF=IN	MACRF=OUT
Base	Opened for input	Opened for in/out
AIXs (UPGRADE=NO)	Not opened	Not opened
AIXs (UPGRADE=YES)	Not opened	Opened for in/out See Note 3 here
PATH (NOUPDATE) to Base	Opened for input No error if already open	Opened for in/out No error if already open
AIXs (UPGRADE=NO)	Not opened	Not opened
AIXs (UPGRADE=YES)	Not opened	Not opened See Note 1 here
PATH (UPDATE) to Base	Opened for input No error if already open	Opened for in/out No error if already open
AIXs (UPGRADE=NO)	Not opened	Not opened
AIXs (UPGRADE=YES)	Not opened	Opened for in/out See Note 3 here
PATH (NOUPDATE) to AIX See Note 4 here	Implied open of Base No error if already open See Note 2 here	Implied open of Base No error if already open See Note 2 here
AIXs (UPGRADE=NO)	AIX opened for input	AIX opened for input
AIXs (UPGRADE=YES)	Not opened See Note 1 here	Not opened See Note 1 here
PATH (UPDATE) to AIX See Note 4 here	Implied open of Base No error if already open See Note 2 here	Implied open of Base No error if already open See Note 2 here
AIXs (UPGRADE=NO)	AIX opened for input	AIX opened for input
AIXs (UPGRADE=YES)	Opened for in/out See Note 3 here	Opened for in/out See Note 3 here

Notes:

1. A NOUPDATE PATH means that the structures for AIXs on the upgrade set are not created
2. The Base is opened by zVSAM for input but has no associated ACB as it hasn't been opened by the app
3. All AIXs on the upgrade set are opened for in/out by zVSAM and may be updated
4. A PATH to an AIX ignores MACRF=IN/OUT

EXLST macro

The EXLST macro will generate an Exit List control block and initialize it according to the parameters specified on the macro invocation

The structure and layout of the generated EXLST are not part of the interface and are therefore not shown in this chapter. Direct access to subfields in the EXLST is discouraged. Use SHOWCB EXLST=, TESTCB EXLST= and/or MODCB EXLST= to inspect, test, and/or modify the EXLST's content

All keywords on the EXLST macro are optional. Before the cluster is opened, all EXLST values can be modified using MODCB EXLST=, or by changing the EXLST directly. The latter is not recommended, as it is not guaranteed to be portable or compatible with future versions of zVSAM

The table below shows how the EXLST macro can be coded:

Opcode	Operand	Remarks
[label] EXLST	[AM=VSAM]	Designates this EXLST as a zVSAM EXLST
	[EODAD=(addr[,mod]])	End-of-data exit routine
	[LERAD=(addr[,mod]])	Logical error analysis routine
	[SYNAD=(addr[,mod]])	Physical error analysis routine
	[JRNAD=(addr[,mod]])	Not supported. Keyword is flagged with a warning message
	[UPAD=(addr[,mod]])	Not supported. Keyword is flagged with a warning message
	[RLSWAIT=(addr[,mod]])	Not supported. Keyword is flagged with a warning message

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

For GENCB MF=I, L or G, a missing addr will generate zero and no error, IBM displays an error
It is assumed that the addr will be made valid by a MODCB EXLST=
A missing mod will generate A

For GENCB MF=E, a missing addr or mod means don't modify that parameter in the CBMR

Note: Although a null address maybe set in the EXLST, you cannot change an address to null with MODCB

EXLST macro parameters

EODAD= Optional parameter to specify the entry address of an exit that handles an end-of-data condition during sequential access
The routine address may be followed by a modifier. For details, please see below
The AMODE for the routine is encoded in the address using the common convention

LERAD= Optional parameter to specify the entry address of an exit that handles logic errors.
The routine address may be followed by a modifier. For details, please see below
The AMODE for the routine is encoded in the address using the common convention

SYNAD= Optional parameter to specify the entry address of an exit that handles physical errors.
The routine address may be followed by a modifier. For details, please see below
The AMODE for the routine is encoded in the address using the common convention

mod modifier, can optionally be specified after each routine address
Values: A or N for Active or Not-active. These are mutually exclusive
As long as the routine is not active it will not be called by zVSAM
The secondary modifier of L (for Load from Linklib) is not supported

Exit logic

This logic is only entered if any of the following conditions are raised:

End-of-data	(EODAD)
Logical error	(LERAD)
Physical error	(SYNAD)

Action	Details
ACBEXLST has an address	No action if zero
Check that the exit is active	No action if inactive
Check that the address is not zero	No action if zero
Branch to the exit address	

CLOSE macro

A cluster needs to be closed after it has been processed. The close macro is used to close one or more clusters and/or one or more sequential files in a single call

Opcode	Operand	Remarks
[label] CLOSE	(entry[,entry]...)	Each cluster or file requires an entry of two parameters
	[MODE=24/31]	Residency mode of all control blocks involved Specify 31 if any reside above the line
	[TYPE=T]	Not supported – future option Keyword is flagged as ignored with a warning message
Entry format:	addr,,	Address of ACB or DCB, followed by two commas to show that options are omitted
	[MF=I or omitted]	Use standard form of CLOSE
	[MF=L]	Use list form of CLOSE
	[MF=(E,addr)]	Use execute form of CLOSE

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

CLOSE macro parameters

For ease of access a short summary follows here:

entry — The CLOSE macro accepts a list of entries. Each entry consists of two consecutive parameters: an address and an optional list of options

address — The address can be specified as an A-type address or as a register. If a register is coded the register number or name must be enclosed in parentheses. The address can be either the address of a DCB or the address of an ACB

options — Code as an omitted parameter

MF=I or omitted

If the MF parameter is omitted a close parmlist is generated inline, plus a call to the CLOSE SVC using the parmlist.

MF=L — With MF=L a close parmlist is generated inline

MF=(E,addr) — Code to modify/populate the close parameter list at the indicated address, which may be a relocatable constant or a (register), plus a call to the CLOSE SVC using the parmlist

CLOSE logic

The close macro generates a close parameter list and/or an SVC 20 instruction to invoke the close routine. The syntax of the close macro is given here

The macro generates the following code:

MF variant	Generated Code
MF=L	Close parameter list data only
MF=(E,address)	1) Code to modify/populate the close parameter list at the indicated address, which may be a relocatable constant or a (register). 2) Code to invoke the close routine
MF=I or omitted	1) Close parameter list data (inline) 2) Code to invoke the close routine

Close parameter list entries have two different formats depending on the MODE parameter.
When MODE=24 then each entry is one fullword
When MODE=31 then each entry is two fullwords

Only one SVC 20 is generated for each CLOSE macro (MF=I or E)
The list format and input to CLOSE depend on MODE=

```
    MODE=24  AL1(option),AL3(DCB/ACB address)
             R1 points to the list

    MODE=31  AL1(option),XL3'00',AL4(DCB/ACB address)
             R0 points to the list and R1=0

    option=0 except for the last entry when option=X'80'
```

CLOSE execution logic

Close involves lock and buffer management and may involve the closure of associated AIXs

CLOSE execution logic is implemented as a Java routine
This logic consists of the following elements:

Action	Details
Determine type of parameter list	31-bit entries, addressed by R0, if R1 = 0 24-bit entries, addressed by R1, if R1 <> 0
loop over all entries in the parameter list	End-of-list is indicated in the option byte of the entry
- check pointer: ACB or DCB	First byte = X'A0' => ACB V1 First four bytes = C'zACB' => ACB V2 First four bytes = C'DCBV' => DCB Otherwise => Error
- if DCB invoke DCB close routine	CLOSE logic for DCB is beyond the scope of this document
- if ACB valid invoke VSAM close routine	
- next entry or end-of-loop	If bit 0 of an entry is on, terminate loop

CLOSE logic for ACB handles a single ACB and proceeds as follows:

Action	Details
Check ACB status	If ACB already closed, issue error and fail close
Check lock status	If any blocks in this dataset or any associated AIX are locked then wait until the locks are freed ???may need a timeout mechanism
Check buffer status	Free any read buffers Write any buffers marked as 'pending write' and then free them
Issue OS close against file	If unsuccessful issue error and fail close

RPL-based interfaces

The RPL is the primary interface for operations at the record level
A program can use multiple RPLs
An RPL must always point to an open ACB in order to specify a valid operation

RPL macro

The RPL macro will generate an RPL and initialize it according to the parameters specified on the macro invocation

Direct access to subfields in the RPL is discouraged. Use SHOWCB RPL=, TESTCB RPL= and/or MODCB RPL= to inspect, test, and/or modify the RPL's content

All keywords on the RPL macro are optional. Before a request is issued, all RPL values can be modified using MODCB RPL=, or by changing the RPL directly. The latter is not recommended, as it is not guaranteed to be portable or compatible with future versions of zVSAM

The table below shows how the RPL macro can be coded

Opcode	Operand	Remarks
[label] RPL	[AM=VSAM]	Designates this RPL as a zVSAM RPL
	[ACB=addr]	Address of an open ACB
	[AREA=addr]	Address of a record area In Move mode the record is read into the area In Locate mode a pointer to the record is moved into the area
	[AREALEN=nr]	Length of record area or record pointer
	[ARG=addr]	Address of the search argument This is a key, a relative record number, or an RBA.
	[ECB=]	Address of an ECB. Used with Asynchronous requests
	[KEYLEN=nr]	Length of key value in ARG when a generic key search is requested
	[MSGAREA=addr]	Address of message area
	[MSGLEN=nr]	Length of message area
	[NXTRPL=addr]	Address of the next RPL in the chain. RPLs can be chained together to request a series ofoperations in a single call to zVSAM
	[OPTCD=(keywd_list)]	List of keywords specifying processing options See table below for valid keywords
	[RECLEN=nr]	Record length. Required when updating or adding records
	[TRANSID=nr]	Not supported – future option. Keyword is flagged as ignored with a warning message

Supported options for the OPTCD parameter are listed below:

Keyword subset	Keyword	Remarks
[ADR | KEY]	ADR	Addressed access to ESDS or KSDS (under review)
	KEY	Keyed access to KSDS or RRDS
	CNV	Not supported – future option. Keyword is flagged as ignored with a warning message
[DIR | SEQ | SKP]	DIR	Direct access to ESDS, KSDS, RRDS
	SEQ	Sequential access to ESDS, KSDS or RRDS
	SKP	Skip sequential access to KSDS or RRDS
[ARD | LRD]	ARD	Access user-defined record location
	LRD	Access last record in the cluster
[FWD | BWD]	FWD	Forward processing
	BWD	Backward processing
[SYN | ASY]	SYN	Synchronous request
	ASY	Asynchronous request
[NUP | UPD | NSP]	NUP	Not for update
	UPD	For update
	NSP	Retain positioning for next sequential access
[KEQ | KGE]	KEQ	Locate record with exact key match
	KGE	Locate record with exact key match, or next higher value
[FKS | GEN]	FKS	Full key search
	GEN	Generic key search. KEYLEN required
[MVE | LOC]	MVE	Move mode
	LOC	Locate mode
[RBA | XRBA]	RBA	4-byte RBA values
	XRBA	8-byte extended RBA values
[NWAITX/WAITX]		Not supported – future option. Keyword is flagged as ignored with a warning message
[CR/NRI]		Not supported – future option. Keyword is flagged as ignored with a warning message

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

POINT macro
GET macro
PUT macro
ERASE macro
CHECK macro
ENDREQ macro
VERIFY macro

GENCB, MODCB, TESTCB and SHOWCB use of the CBMR

A CBMR is generated for all forms of these macros

Direct access to subfields in the CBMR is discouraged. Use SHOWCB, TESTCB and/or MODCB to inspect, test, and/or modify the content of an ACB, EXLST, or RPL. Use the appropriate MF= parameter on any of these macros to modify and/or use a CBMR

The CBMR consists of three parts: a header, a body, and a tail. The header has a fixed layout. The body consists of request-dependent fields and a list of verb codes. The tail contains all the data fields that go with the verb codes. Data fields can be 0, 4 or 8 bytes in length.
Verb codes X'01'-X'5F' have a zero-length data field (i.e. no data field)
Verb codes X'60'-X'DF' have a 4-byte data field
Verb codes X'E0'-X'FF' have an 8-byte data field
All data fields in the tail are allocated consecutively, in the same order as the verbs that define their meaning

CBMR – header

The CBMR header identifies the type (ACB, EXLST, RPL, GENCB, MODCB, SHOWCB or TESTCB)
It also has details of any work area needed and a count of verbs in CBMRVRBS

CBMR – body

It's length is determined by the CBMRVRBS fields in the CBMR header
It contains one verb code for each specified parameter

CBMR – tail

The body is directly followed by the tail
It contains a data field of 4 or 8 bytes for each verb coded in the body, in the same sequence
The starting point of the tail can be found by adding the CBMRVRBS value to the end of the CBMR header
Its length can be calculated from the CBMRSIZE field, by subtracting both the header length and the CBMRVRBS field

Additional notes:
CBMRACB_NRS – TESTCB only, always true
CBMRACB_RST – TESTCB only, always false
CBMRACB_NSR – TESTCB only, always true
CBMRACB_LSR – TESTCB only, always false
CBMRACB_GSR – TESTCB only, always false
CBMRACB_RLS – TESTCB only, always false
CBMRACB_NUB – TESTCB only, always true
CBMRACB_UBF – TESTCB only, always false
CBMRACB_CFX – TESTCB only, always false
CBMRACB_NFX – TESTCB only, always false
CBMRACB_DDN – TESTCB only, always false
CBMRACB_DSN – TESTCB only, always false
CBMRACB_ICI – TESTCB only, always false
CBMRACB_NCI – TESTCB only, always true
CBMRACB_LEW – TESTCB only, always true
CBMRACB_NLW – TESTCB only, always false
CBMRACB_REPL – TESTCB only, always false
CBMRACB_SSWD – TESTCB only, always false
CBMRACB_WCK – TESTCB only, always false
CBMRACB_CMPRS – TESTCB only, always false
CBMRACB_XADDR – TESTCB only, always true

CBMRACB_COPIES – GENCB only. If not specified, the CBMR handler assumes 1.
CBMRACB_PASSWD – pointer to a one-byte length field, followed by the password
CBMRACB_MAREA – TESTCB only, always 0
CBMRACB_MLEN – TESTCB only, always 0
CBMRACB_SHRPL – TESTCB only, always 0
CBMRACB_ENDRBA – ending RBA of the component, derived from ending XLRA.
CBMRACB_FS – Nr of free blocks per 100
CBMRACB_HALCRBA – High allocated RBA, derived from highest allocated XLRA.
CBMRACB_NEXT – for zVSAM the value is always 1.
CBMRACB_NSSS – always 0
CBMRACB_LEVEL – 4-byte address followed by 4-byte length of level info field
CBMRACB_LOKEY – 4-byte address followed by 4-byte length of key field
CBMRACB_RELEASE – 4-byte address followed by 4-byte length of level info field

Additional notes:
CBMRRPL_CNV – TESTCB only, always false

GENCB, MODCB, TESTCB and SHOWCB use of MF=

MF=I or omitted   Generates CBMR and invokes ZVSAM19C to retrieve fields

MF=L   Generates CBMR inline
MF=(L,addr)   Generates CBMR inline and then moves it to addr
MF=(L,addr,label)   as above and generates label equ size

MF=(E,addr)   Modifies the CBMR at addr
                    Invokes ZVSAM19C to retrieve fields using the CBMR

MF=(G,addr)   Generates CBMR inline and then moves it to addr
                    Invokes ZVSAM19C to retrieve fields using the CBMR
MF=(G,addr,label) as above and generates label equ size

addr can be label or reg, reg cannot be R0, R1, R14 or R15
reg is not permitted for MF=L

GENCB BLK=ACB macro

This GENCB macro will generate ACBs and initialize or change them according to the parameters specified on the macro invocation. It is for this reason that all supported parameters and keywords of the ACB macro (as described above) are supported on the GENCB macro

Direct access to subfields in the ACB is discouraged. Use SHOWCB ACB=, TESTCB ACB= and/or MODCB ACB= to inspect, test, and/or modify the ACB's content

Direct access to subfields in the CBMR is strongly discouraged

The GENCB ACB macro can be coded as follows:

Opcode	Operand	Remarks
[label] GENCB	BLK=ACB	Instructs GENCB to generate 1 or more ACBs
	[AM=VSAM]	Optional, no other values allowed
	[COPIES=1]	The number of identical ACBs to generate Specify a number between 1 and 65535
	[WAREA=addr]	The work area where the ACBs are to be constructed
	[LENGTH=nr]	Length of the work area in bytes If WAREA/LENGTH are omitted then storage is dynamically acquired
	[LOC=BELOW | ANY]	Where GENCB is to allocate dynamically acquired storage if needed
	[other]	**Any parameter supported on the ACB macro**
	[MF=]	See the description of MF= here

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

WAREA= When WAREA is specified, LENGTH must be specified too
When WAREA is not specified, the CBMR handler allocates an area of storage
The address of this area is returned in R1; its length in R0

LENGTH= Length in bytes of the area indicated by WAREA
When LENGTH is specified, WAREA must be specified as well

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=9	WAREA is too small
R15=8	Reason Code=n/a	Invalid CBMR An attempt was made to update a CBMR with a field not previously created

GENCB BLK=EXLST macro

The GENCB macro with BLK=EXLST will generate or manipulate Exit Lists for use with ACBs and initialize or change them according to the parameters specified on the macro invocation. It is for this reason that all supported parameters and keywords of the EXLST macro (as described above) are supported on the GENCB macro when BLK=EXLST is specified

Direct access to subfields in the EXLST is discouraged. Use SHOWCB EXLST=, TESTCB EXLST= and/or MODCB EXLST= to inspect, test, and/or modify the EXLST's content

Direct access to subfields in the CBMR is strongly discouraged

The GENCB EXLST macro can be coded as follows:

Opcode	Operand	Remarks
[label] GENCB	BLK=EXLST	Instructs GENCB to generate one or more EXLSTs
	[AM=VSAM]	Optional, no other values allowed
	[COPIES=1]	The number of identical EXLSTs to generate Specify a number between 1 and 65535
	[WAREA=addr]	The work area where the EXLSTs are to be constructed
	[LENGTH=nr]	Length of the work area in bytes If WAREA/LENGTH are omitted then storage is dynamically acquired
	[LOC=BELOW | ANY]	Where GENCB is to allocate dynamically acquired storage if needed
	[other]	Any parameter supported on the EXLST macro
	[MF=]	See the description of MF= here

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

WAREA= When WAREA is specified, LENGTH must be specified too
When WAREA is not specified, the CBMR handler allocates an area of storage
The address of this area is returned in R1; its length in R0

LENGTH= Length in bytes of the area indicated by WAREA.
When LENGTH is specified, WAREA must be specified as well

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=9	WAREA is too small
R15=8	Reason Code=n/a	Invalid CBMR An attempt was made to update a CBMR with a field not previously created

GENCB BLK=RPL macro

The GENCB BLK=RPL macro generates or manipulates RPLs and initializes or changes them according to the parameters specified on the macro invocation. It is for this reason that all supported parameters and keywords of the RPL macro (as described above) are supported on the GENCB macro

Direct access to subfields in the RPL is discouraged. Use SHOWCB RPL=, TESTCB RPL= and/or MODCB RPL= to inspect, test, and/or modify the RPL's content

Direct access to subfields in the CBMR is strongly discouraged

The GENCB RPL macro can be coded as follows:

Opcode	Operand	Remarks
[label] GENCB	BLK=RPL	Instructs GENCB to generate 1 or more RPLs
	[AM=VSAM]	Optional, no other values allowed
	[COPIES=1]	The number of identical RPLs to generate Specify a number between 1 and 65535
	[WAREA=addr]	The work area where the RPLs are to be constructed
	[LENGTH=nr]	Length of the work area in bytes If WAREA/LENGTH are omitted then storage is dynamically acquired
	[LOC=BELOW | ANY]	Where GENCB is to allocate dynamically acquired storage - if needed
	[other]	**Any parameter supported on the RPL macro**
	[MF=]	See the description of MF= here

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

WAREA= When WAREA is specified, LENGTH must be specified too
When WAREA is not specified, the CBMR handler allocates an area of storage
The address of this area is returned in R1; its length in R0

LENGTH= Length in bytes of the area indicated by WAREA.
When LENGTH is specified, WAREA must be specified as well

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=9	WAREA is too small
R15=8	Reason Code=n/a	Invalid CBMR An attempt was made to update a CBMR with a field not previously created

MODCB ACB= macro

The MODCB macro with ACB=addr will modify an ACB according to the parameters specified on the macro invocation. It is for this reason that all parameters and keywords of the ACB macro (as described above) are supported on the MODCB macro when ACB=addr is specified

Direct access to subfields in the ACB is discouraged. Use SHOWCB ACB=, TESTCB ACB= and/or MODCB ACB= to inspect, test, and/or modify the ACB's content. See note here

Direct access to subfields in the CBMR is strongly discouraged

The MODCB ACB macro can be coded as follows:

Opcode	Operand	Remarks
[label] MODCB	ACB=address	Points MODCB to the ACB to be modified
	[AM=VSAM]	Optional, no other values allowed
	[other]	**Any parameter supported on the ACB macro**
	[MF=]	See the description of MF= here

All supported parameters are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual.

Note: When an ACB has MACRF=(OUT) which allows read and write functions it is not possible to change the ACB to read-only using MODCB
If this is needed code the instruction NI ACBMACR1,255-ACBOUT

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=4	ACB= does not point to an ACB
R15=4	Reason Code=12	MODCB was attempted on an open ACB
R15=8	Reason Code=n/a	Invalid CBMR or ACB
		An attempt was made to update a CBMR with a field not previously created

MODCB EXLST= macro

The MODCB macro with EXLST=addr will modify an EXLST according to the parameters specified on the macro invocation. It is for this reason that all parameters and keywords of the EXLST macro (as described above) are supported on the MODCB macro when EXLST=addr is specified

Direct access to subfields in the EXLST is discouraged. Use SHOWCB EXLST=, TESTCB EXLST= and/or MODCB EXLST= to inspect, test, and/or modify the EXLST's content

Direct access to subfields in the CBMR is strongly discouraged

The MODCB EXLST macro can be coded as follows:

Opcode	**Operand**	**Remarks**
[label] MODCB	EXLST=addr	Points MODCB to the EXLST to be modified
	[AM=VSAM]	Optional, no other values allowed
	[other]	Any parameter supported on the EXLST macro
	[MF=]	See the description of MF= here

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=4	EXLST= does not point to an EXLST
R15=8	Reason Code=n/a	Invalid CBMR or EXLST An attempt was made to update a CBMR with a field not previously created

MODCB RPL= macro

The MODCB macro with RPL=addr will modify an RPL according to the parameters specified on the macro invocation. It is for this reason that all parameters and keywords of the RPL macro (as described above) are supported on the MODCB macro when RPL=addr is specified

Direct access to subfields in the RPL is discouraged. Use SHOWCB RPL=, TESTCB RPL= and/or MODCB RPL= to inspect, test, and/or modify the RPL's content

Direct access to subfields in the CBMR is strongly discouraged

The MODCB RPL macro can be coded as follows:

Opcode	Operand	Remarks
[label] MODCB	RPL=addr	Points MODCB to the RPL to be modified
	[AM=VSAM]	Optional, no other values allowed
	[other]	**Any parameter supported on the RPL macro**
	[MF=]	See the description of MF= here

All supported parameters are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=4	RPL= does not point to an RPL
R15=8	Reason Code=n/a	Invalid CBMR or RPL
		An attempt was made to update a CBMR with a field not previously created

SHOWCB with no specified block type macro

The SHOWCB macro without a block will return length fields according to the parameters specified on the macro invocation in the order they are specified. Duplicates are permitted

Opcode	Operand	Remarks
[label] SHOWCB	[AM=VSAM]	Optional, no other values allowed
	AREA=addr	Address of return area
	LENGTH=nr	Size of return area in bytes
	FIELDS=(keywd_list)	List of keywords indicating which fields to return
4	[MF=]	See the description of MF= here

Supported options for the FIELDS parameter are listed below:

Keyword	Length	Remarks
ACBLEN	4	Length of ACB in bytes
EXLLEN	4	Length of EXLST in bytes
RPLLEN	4	Length of RPL in bytes

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation For details, please refer to the relevant IBM manual

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=9	Length too small
R15=8	Reason Code=n/a	Invalid CBMR An attempt was made to update a CBMR with a field not previously created

SHOWCB ACB= macro

The SHOWCB macro with ACB=addr will return ACB-related fields according to the parameters specified on the macro invocation in the order they are specified. Duplicates are permitted

Direct access to subfields in the ACB is discouraged. Use SHOWCB ACB=, TESTCB ACB= and/or MODCB ACB= to inspect, test, and/or modify the ACB's content

Direct access to subfields in the CBMR is strongly discouraged

The SHOWCB ACB macro can be coded as follows:

Opcode	Operand	Remarks
[label] SHOWCB	ACB=address	Points SHOWCB to the ACB to be queried
	[AM=VSAM]	Optional, no other values allowed
	AREA=addr	Address of return area
	LENGTH=nr	Size of return area in bytes
	[OBJECT=DATA/INDEX]	For KSDS: select data or index component
	FIELDS=(keywd_list)	List of keywords indicating which fields to return
	[MF=]	See the description of MF= here

Supported options for the FIELDS parameter are listed below:

Keyword	Length	Remarks
ACBLEN	4	Length of ACB in bytes
AVSPAC	4	Available space in data/index (last 4 bytes). Derived from CTRAVSPAC
BFRFND	4	Nr of buffer hits for data/index including LSR (last 4 bytes) Derived from CTRNBFRFND
BSTRNO	4	Initial nr of strings for AIX. Derived from ACBBSTNO
BUFND	4	Nr of data buffers specified in ACB. Derived from ACBBUFND
BUFNI	4	Nr of index buffers specified in ACB. Derived from ACBBUFNI
BUFNO	4	Number of data/index buffers allocated (last 4 bytes) Derived from CTRNBUFNO
BUFNOL	4	Number of data/index buffers allocated for LSR processing (returns zero)
BUFRDS	4	Number of data/index buffer reads. Derived from CTRNBUFRDS
BUFSP	4	Buffer space in bytes specified in ACB. Derived from ACBBUFSP
BUFUSE	4	Number of data/index buffers actually in use. Derived from CTRNBUFUSE
CDTASIZE	8	Size of a compressed dataset (returns zero)
CINV	4	Block size for data/index. Derived from PFXBLKSZ
CIPCA	4	CI's in CA (returns zero)
DDNAME	8	DDNAME specified in ACB. Derived from ACBDDNM
ENDRBA	4	Highest used RBA. Derived from CTRENDRBA (last 4 bytes)
ERROR	4	Return code from last open/close operation. Derived from ACBERFLG
EXLLEN	4	Length of EXLST in bytes

Keyword	Length	Remarks
EXLST	4	Ptr to EXLST, zero if none. Derived from ACBEXLST
FS	4	Nr of data free CIs per CA (returns zero)
HALCRBA	4	Highest allocated data/index RBA Derived from CTRHALCRBA (last 4 bytes)
HLRBA	4	For OBJECT=INDEX only, highest index block RBA Derived from CTRHLRBA
KEYLEN	4	Length of key field. Derived from PFXKYLEN
LEVEL	8	Address (4 bytes) and length (4 bytes) of field containing zVSAM version Derived from ACBVER
LOKEY	8	Address (4 bytes) of lowest key in the cluster + length (4 bytes) of key Derived from CTRLOKEY@ and PFXKYLEN
LRECL	4	Maximum data/index record length. Derived from PFXRCLEN
MAREA	4	Message area (returns foxes)
MLEN	4	Message length (returns zero)
NCIS	4	Nr of Block splits in the data component. Zero for OBJECT=INDEX. Derived from CTRNCIS (last 4 bytes)
NDELR	4	Nr of deleted records from the data component (last 4 bytes) Zero for OBJECT=INDEX. Derived from CTRNDELR
NEXCP	4	Nr of I/O requests for the data/index components (last 4 bytes) Derived from CTRNEXCP
NEXT	4	Nr of extents of the data/index components (returns 1)
NINSR	4	Nr of records inserted for the data component (last 4 bytes) Zero for OBJECT=INDEX. Derived from CTRNINSR
NIXL	4	Nr of index levels for index component. Zero for OBJECT=DATA Derived from highest non-foxes PFXBLVLn
NLOGR	4	Nr of records in the data/index (last 4 bytes). Derived from CTRNLOGR
NRETR	4	Nr of records retrieved from the data component (last 4 bytes). Zero for OBJECT=INDEX. Derived from CTRNRETR
NSSS	4	Nr of control area splits for the data/index (returns zero)
NUIW	4	Nr of non-user writes (last 4 bytes). Derived from CTRNNUIW
NUPDR	4	Nr of updated records in the data/index components (last 4 bytes). Derived from CTRNUPDR
PASSWD	4	Ptr to password, consisting of length (1 byte, binary) followed by the actual password value. Derived from ACBPASSW
RELEASE	8	Address (4 bytes) and length (4 bytes) of field containing zVSAM version Derived from ACBVER. Same as LEVEL
RKP	4	Relative Key Position, offset of key within logical record Derived from PFXKYOFF
RMODE31	4	0=None, 1=Buff, 2=CB, 3=All. Derived from ACBOFLGS
RPLLEN	4	Length of RPL in bytes

Keyword	Length	Remarks
SDTASIZE	8	Uncompressed data size. Derived from CTRSDTASZ
SHRPOOL	4	SHRPOOL number. Derived from ACBSHRP
STMST	8	STCK of last close. Derived from CTRSTMST
STRMAX	4	Max nr of concurrently active strings (last 4 bytes). Derived from CTRSTRMAX
STRNO	4	Max nr of allocated strings. Derived from ACBSTRNO
UIW	4	Nr of user writes for data/index (last 4 bytes). Derived from CTRNUIW
XAVCSPAC	8	AVCSPAC when value may exceed 4GB
XBFRFND	8	BFRFND when value may exceed 4GB
XBUFNO	8	BUFNO when value may exceed 4GB
XBUFUSE	8	BUFUSE when value may exceed 4GB
XBUFRDS	8	BUFRDS when value may exceed 4GB
XENDRBA	8	ENDRBA when value may exceed 4GB
XHALCRBA	8	HALCRBA when value may exceed 4GB
XHLRBA	8	HLRBA when value may exceed 4GB
XNCIS	8	NCIS when value may exceed 4GB
XNDELR	8	NDELR when value may exceed 4GB
XNEXCP	8	NEXCP when value may exceed 4GB
XNINSR	8	NINSR when value may exceed 4GB
XNLOGR	8	NLOGR when value may exceed 4GB
XNRETR	8	NRETR when value may exceed 4GB
XNUIW	8	NNUIW when value may exceed 4GB
XSTRMAX	8	STRMAX when value may exceed 4GB
XUIW	8	UIW when value may exceed 4GB

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=1	ACBPFX or ACBXPFX are zero (X)HLRBA requested and OBJECT=DATA For fields that have 8-byte values (eg. XHLRBA) the 4-byte version is requested but the 1st four bytes are not zero CTRLOKEY@ is foxes for: non-KSDS KSDS index KSDS data but empty
R15=4	Reason Code=9	Length too small
R15=8	Reason Code=n/a	Invalid CBMR or ACB An attempt was made to update a CBMR with a field not previously created

SHOWCB EXLST= macro

The SHOWCB macro with EXLST=addr will return EXLST-related fields according to the parameters specified on the macro invocation in the order they are specified. Duplicates are permitted

Direct access to subfields in the EXLST is discouraged. Use SHOWCB EXLST=, TESTCB= EXLST and/or MODCB EXLST= to inspect, test, and/or modify the EXLST's content

Direct access to subfields in the CBMR is strongly discouraged

The SHOWCB EXLST= macro can be coded as follows:

Opcode	Operand	Remarks
[label] SHOWCB	EXLST=addr	Points SHOWCB to the EXLST to be queried
	[AM=VSAM]	Optional, no other values allowed
	AREA=addr	Address of return area
	LENGTH=nr	Size of return area in bytes
	FIELDS=(keywd_list)	List of keywords indicating which fields to return
	[MF=]	See the description of MF= here

Supported options for the FIELDS parameter are listed below:

Keyword	Length	Remarks
ACBLEN	4	Length of ACB in bytes
EODAD	4	End-of-data exit routine address
EXLLEN	4	Length of EXLST in bytes
JRNAD	4	Supported here, but as it's not supported by other macros, zero is returned
LERAD	4	Logical error analysis routine address
RPLLEN	4	Length of RPL in bytes
SYNAD	4	Physical error analysis routine address

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=9	Length too small
R15=8	Reason Code=n/a	Invalid CBMR or EXLST An attempt was made to update a CBMR with a field not previously created

SHOWCB RPL= macro

The SHOWCB macro with RPL=addr will return RPL-related fields according to the parameters specified on the macro invocation in the order they are specified. Duplicates are permitted

Direct access to subfields in the RPL is discouraged. Use SHOWCB RPL=, TESTCB RPL= and/or MODCB RPL= to inspect, test, and/or modify the RPL's content

Direct access to subfields in the CBMR is strongly discouraged

The SHOWCB RPL= macro can be coded as follows:

Opcode	Operand	Remarks
[label] SHOWCB	RPL=addr	Points SHOWCB to the RPL to be queried
	[AM=VSAM]	Optional, no other values allowed
	AREA=addr	Address of return area
	LENGTH=nr	Size of return area in bytes
	FIELDS=(keywd_list)	List of keywords indicating which fields to return
	[MF=]	See the description of MF= here

Supported options for the FIELDS parameter are listed below:

Keyword	Length	Remarks
ACB	4	Pointer to ACB
ACBLEN	4	Length of ACB in bytes
AIXPC	4	Alternate index pointer count. Derived from PFXAIXN
AREA	4	Pointer to record buffer
AREALEN	4	Size of record buffer in bytes
ARG	4	Pointer to last used search argument field
ECB	4	Pointer to user-supplied ECB
EXLLEN	4	Length of EXLST in bytes
FDBK	4	Feedback code for the last request
FTNCD	4	Function code
KEYLEN	4	Length of key, for use with OPTCD=GEN
MSGAREA	4	Pointer to message area (returns foxes)
MSGLEN	4	Length of message area (returns zero)
NXTRPL	4	Pointer to next RPL, if any
RBA	4	4-byte RBA of last record processed (ESDS ony, otherwise zero)
RECLEN	4	Length of current record
RPLLEN	4	Length of RPL in bytes
TRANSID	4	Transaction id (returns foxes)
XRBA	8	8-byte RBA of last record processed (ESDS only, otherwise zero)

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation
For details, please refer to the relevant IBM manual

Return (R15) and Reason (R0) Codes:

R15=0	Reason Code=n/a	Successful
R15=4	Reason Code=1	AIXPC or RPLDACB are zero
R15=4	Reason Code=9	Length too small
R15=8	Reason Code=n/a	Invalid CBMR or RPL An attempt was made to update a CBMR with a field not previously created

TESTCB ACB= macro

The TESTCB macro with ACB=addr will test ACB-related fields according to the parameters specified on the macro invocation. Only a single test can be specified on each TESTCB invocation. TESTCB returns a PSW condition code of 8=Equal when the specified test is met, 7=NotEqual otherwise.

Direct access to subfields in the ACB is discouraged. Use SHOWCB ACB=, TESTCB ACB= and/or MODCB ACB= to inspect, test, and/or modify the ACB's content.

Direct access to subfields in the CBMR is strongly discouraged

The TESTCB ACB macro can be coded as follows:

Opcode	Operand	Remarks
[label] TESTCB	ACB=address	Points TESTCB to the ACB to be tested
	[AM=VSAM]	Optional, no other values allowed
	ERET=addr	Address of error handling routine
	[OBJECT=DATA/INDEX]	For KSDS: select data or index component

	ATRB=(keywd_list)
	ATRB=COMPRESS
	ATRB=UNQ
	ATRB=XADDR
		List of keywords indicating attributes to test
		Compression on? Always false for zVSAM.
		Path is defined on unique key?
		Extended format? Always true for zVSAM.

	OFLAGS=OPEN	Opened successfully?
	OPENOBJ=PATH/BASE/AIX	ACB represents Path/Base/AIX?
	ACBLEN=nr	length of ACB in bytes
	AVSPAC=nr	available space in bytes
	BSTRNO=nr	Initial nr of strings
	BUFND=nr	Nr of data buffers
	BUFNI=nr	Nr of index buffers
	BUFNO=nr	nr of I/O Buffers
	BUFSP=nr	Buffer space in bytes
	CINV=nr	Control interval size / Block size in bytes
	DDNAME=string	DDNAME
	ENDRBA=nr	High water mark XLRA
	ERROR=nr	Error code of last error
	EXLST=adr	EXLST address
	FS=nr	Free Block per 100
	KEYLEN=nr	Length of key field
	LRECL=nr	Logical Record Length
	MAREA=adr	Message area address
	MLEN=nr	Length of message area in bytes
	NCIS=nr	Nr of Block splits
	NDELR=nr	Nr of deleted records

Opcode	Operand	Remarks
	NEXCP=nr	Nr of I/O requests
	NEXT=nr	Nr of extents
	NINSR=nr	Nr of records inserted
	NIXL=nr	Nr of index levels
	NLOGR=nr	Nr of records
	NRETR=nr	Nr of records retrieved
	NSSS=nr	Nr of control area splits. Foxes.
	NUPDR=nr	Nr of updates applied
	PASSWD=adr	Ptr to 1-byte length followed by password
	RKP=nr	Offset of key field within record
	SHRPOOL=nr	SHRPOOL number
	STMST=adr	Poijnter to system timestamp field
	STRNO=nr	Max. nr of parallel requests
	[MF=I or omitted]	Use standard form of TESTCB ACB
	[MF=(L[,addr][,label])]	Use list form of TESTCB ACB
	[MF=(E,addr)]	Use execute form of TESTCB ACB
	[MF=(G,addr,[label])]	Use generate form of TESTCB ACB

Supported options for the ATRB parameter are listed below:

Keyword	Remarks
ESDS	Component is an ESDS?
KSDS	Component is a KSDS?
LDS	Component is an LDS?
RRDS	Component is a RRDS?
REPL	Always false for zVSAM.
SPAN	Component may hold segmented records
SSWD	Always false for zVSAM.
VRRDS	Variable-length RRDS?
VESDS	Variable-length ESDS? (zVSAM extension)
WCK	Always false for zVSAM.

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual.

For ease of access a short summary can be found in the addenda here

TESTCB EXLST= macro

The TESTCB macro with EXLST=addr will test EXLST-related fields according to the parameters specified on the macro invocation. Only a single test can be specified on each TESTCB invocation
TESTCB returns a PSW condition code

Direct access to subfields in the EXLST is discouraged. Use SHOWCB EXLST=, TESTCB EXLST= and/or MODCB EXLST= to inspect, test, and/or modify the EXLST's content.

Direct access to subfields in the CBMR is strongly discouraged.

The TESTCB EXLST macro can be coded as follows:

Opcode	Operand	Remarks
[label] TESTCB	EXLST=addr	Points TESTCB to the EXLST to be tested
	[AM=VSAM]	Optional, no other values allowed
	ERET=addr	Address of error handling routine

	EODAD=0
	EODAD=addr[,mod]
		End-of-data exit routine address

	JRNAD=0
	JRNAD=addr[,mod]
		Not supported. Keyword is flagged with a warning message

	UPAD=0
	UPAD=addr[,mod]
		Not supported. Keyword is flagged with a warning message

	RLSWAIT=0
	RLSWAIT=addr[,mod]
		Not supported. Keyword is flagged with a warning message

	LERAD=0
	LERAD=addr[,mod]
		Logical error analysis routine address

	SYNAD=0
	SYNAD=addr[,mod]
		Physical error analysis routine address

	EXLLEN=nr	Size of EXLST in bytes
	[MF=I or omitted]	Use standard form of TESTCB EXLST
	[MF=(L[,addr][,label])]	Use list form of TESTCB EXLST
	[MF=(E,addr)]	Use execute form of TESTCB EXLST
	[MF=(G,addr,[label])]	Use generate form of TESTCB EXLST

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual

TESTCB RPL= macro

The TESTCB macro with RPL=addr will test RPL-related fields according to the parameters specified on the macro invocation. Only a single test can be specified on each TESTCB invocation. TESTCB returns a PSW condition code of 8=Equal when the specified test is met, 7=NotEqual otherwise.

Direct access to subfields in the RPL is discouraged. Use SHOWCB RPL, TESTCB RPL and/or MODCB RPL to inspect, test, and/or modify the RPL's content.

Direct access to subfields in the CBMR is strongly discouraged.

The TESTCB RPL macro can be coded as follows:

Opcode	Operand	Remarks
[label] TESTCB	RPL=addr	Points TESTCB to the RPL to be tested
	[AM=VSAM]	Optional, no other values allowed
	ERET=addr	Address of error handling routine
	OPTCD=(keywd_list)	List of keywords indicating attributes to test
	AIXFLAG=AIXPKP	Using primary keys
	AIXPC=nr	Nr of index pointers in use
	FTNCD=nr	Reflects the condition of the upgrade set
	IO=COMPLETE	
	ACB=addr	
	AREA=addr	
	AREALEN=addr	
	ARG=addr	
	ECB=addr	
	FDBK=nr	
	KEYLEN=nr	Length of key field
	RECLEN=nr	Logical Record Length
	MSGAREA=adr	Message area address
	MSGLEN=nr	Length of message area in bytes
	NXTRPL=addr	
	RBA=nr	
	RPLLEN=nr	
	TRANSID=nr	
	[MF=I or omitted]	Use standard form of TESTCB RPL
	[MF=(L[,addr][,label])]	Use list form of TESTCB RPL
	[MF=(E,addr)]	Use execute form of TESTCB RPL
	[MF=(G,addr,[label])]	Use generate form of TESTCB RPL

Supported options for the OPTCD parameter are the same as those available on the RPL macro

All supported parameters and keywords are implemented compatibly with IBM's VSAM implementation. For details, please refer to the relevant IBM manual

Overview of differences with IBM VSAM:

RBA=nr – zVSAM supports this keyword only for ESDS. For any other type of cluster a value of foxes will be assumed by default

Catalog management

This is where all meta-data about the zVSAM components are kept and where the relations between zVSAM components are defined. Catalogs are currently created as static assembled modules
Extended catalogs contained in datasets will be considered in a future release

The catalog will hold at least:
- file name
- pointer to index file
- pointers to all related AIX clusters
- LRECL
- record type (F, V, FS, VS)
- type of component (ESDS, KSDS, RRDS, AIX)
- freeblocks (during load, between blocks)
- freespace (during load, within blocks)
- Physical Block size (aka CI-size, 512 bytes to 16MB)

For a complete list of catalog components please see the “z390_zVSAM_Catalog_User_Guide”

Physical structure of the files

Basic Concepts

Files, Blocks, Records

The logical unit of access or storage is the record. Yet the unit for any given I/O operation is the block. Block sizes may vary from 512 bytes to 16MB. Each block holds up to 255 records. For any given cluster component, choosing an appropriate block size is important. Block size can greatly affect not only performance, but also both internal and external storage consumption

A cluster consists of one or more files that belong together and should be managed together. Whether you take a backup, perform a restore, or perform other administrative tasks, the files that make up a cluster should be managed alike. When creating a backup copy of a cluster or restoring a cluster, make sure no other processes try to access the data at the same time

zVSAM implements a number of checks and balances to prevent inadvertent access to data that may have been compromised. Names and locations of files are managed. Tampering with files or file attributes may render the cluster unusable

As a result, it is not possible to rename a zVSAM cluster or file. Unload and reload your cluster in order to move the data or to assign a different name to cluster or file

Just like files in a cluster belong together and should be managed together, clusters in a sphere are logically connected and should be managed together. Again, failing to manage the files in a correct and comprehensive manner may render your data inaccessible

Cluster types and Cluster Components

Each cluster consists of a data component and an index component as follows:

Data and Index Components per Cluster type

Cluster type	**Index content**
ESDS	Index on XRBA
KSDS	Index on key value
RRDS	Index on RRN
AIX	Index on alternate key value

Record Formats

In zVSAM we support the following record formats:

Record Formats	
Format	**Properties**
F	Fixed. All records have the same length Records never span a Block boundary
FS	Fixed Spanned. All records have the same length Records expected to span a Block boundary
V	Variable. Records have varying lengths Records never span a Block boundary
VS	Variable Spanned. Records have varying lengths Records may or may not span a Block boundary

For ESDS, KSDS, and RRDS all record types are supported.
For AIX only F and VS record formats are supported: F for unique, and VS for non-unique indexes

Supported Record Formats				
Cluster Type	**F**	**FS**	**V**	**VS**
ESDS	Y	Y	Y*	Y*
KSDS	Y	Y	Y	Y
RRDS	Y	Y*	Y	Y*
AIX - unique	Y	N	N	N
AIX - non-unique	N	N	N	Y
*** zVSAM extension**				

For a unique AIX each record holds an alternate key value plus the primary key (KSDS) or XLRA (ESDS) of the associated record in the cluster's data component. This fixed configuration dictates a record type of F

For a non-unique AIX each record holds an alternate key value and as many primary keys (KSDS) or XLRAs (ESDS) of associated records in the cluster's data component as there are records holding that specific alternate key value. The table of primary keys may vary in length from 1 to very large numbers. No block size is guaranteed to be large enough to hold the largest possible index record, therefore a record type of VS is mandated. When a non-unique index record needs to be split into segments, no primary key value or XLRA is ever split; i.e. only an exact number of these reside within a single segment of the record

Supported Index-types			
Cluster Type	**Primary - Unique**	**AIX - unique**	**AIX - Non-unique**
ESDS	Y*	Y	Y
KSDS	Y	Y	Y
RRDS	Y*	N	N
*** zVSAM extension**			

File Structure

Physical files

All zVSAM data is stored in physical files, as defined to the operating system
Each component consists of one file. This file is formatted as a zVSAM file, the structure of which is explained in the next set of chapters

Please note: the hosting operating system may impose a limit on physical file size and not every host OS supports a physical file spanning a volume boundary of the storage device(s). Therefore, to support clusters that exceed the maximum size of a single physical file, in the future we may need to support clusters that consist of multiple files

Structure of physical files

Every zVSAM file has a block size. The block being the basic unit of I/O. The first block of every file is the prefix block, which is always 4096 bytes in size. The prefix block holds information about the cluster, its data, and its structure
Data in the prefix block are not accessible to user programs. However, selected fields in the prefix block can be queried using a SHOWCB ACB= request

All Data and Index blocks in the file have a user-defined blocksize (DATABLOCKSIZE= and INDEXBLOCKSIZE=). The file is assumed to logically begin with the first block after the prefix block
There are 5 types of blocks that may occur in zVSAM files:

1) Prefix block – one for each file, being the first 4096 bytes of every file
2) Spacemap block – used to manage free space in the file
3) Data block – used to hold user data, or AIX data records (in an AIX only)
4) Index block – used to hold index information
5) ELIX block – used to index segmented (read: large) non-unique AIX records

Every block has an internal structure consisting of a block header, a list of record pointers, a block body and a block footer. The block header and footer have a fixed structure. The list of record pointers has a variable length. The block body contains record data and/or free space
Each of the 5 block types is explained in more detail below

Block Header Structure

Every block has a block header (ZVSAMHDR). All block headers have the same structure

BHDRSEQ# is incremented by one every time the block is written out to the file
The footer area contains a comparable field: BFTRSEQ#. Together they guard against incomplete writes

BHDRXLVL indicates the index level. Zero is the leaf level. Index blocks are chained by level. That is, for every index level in use there is a pair of pointers in the prefix block (PFXBLVLn/PFXELVLn) that starts and ends the chain for that level

BHDRSELF contains the block's own XLRA. This helps to guard against misdirected reads and/or writes.
BHDRNEXT/BHDRPREV point to the next and previous block on the chain. Which chain this is, depends on the BHDRFLAG setting, and, if this is an index block, by the BHDRXLVL value
For the prefix block, these two fields are set to foxes

Segmented records are a special case. Segments of a segmented record never share their block with other data. The block holding the first segment is part of the data chain. A block holding a non-first segment is part of the segment chain. A block that holds a record's first segment has an SPX pointing to the block holding the next segment. Subsequent segments are retrieved by following the SPXs to the last segment of the record

The Segment chain starting at PFXBSEGM and ending at PFXESEGM has no role in processing a spanned dataset but just provides an extra integrity check

Block Footer Structure

Every block has a block footer zVSAMFTR). All block footers have the same structure

BFTRSEQ# is incremented by one every time the block is written out to the file
The header area contains a comparable field: BHDRSEQ#. Together they guard against incomplete writes

Prefix Block

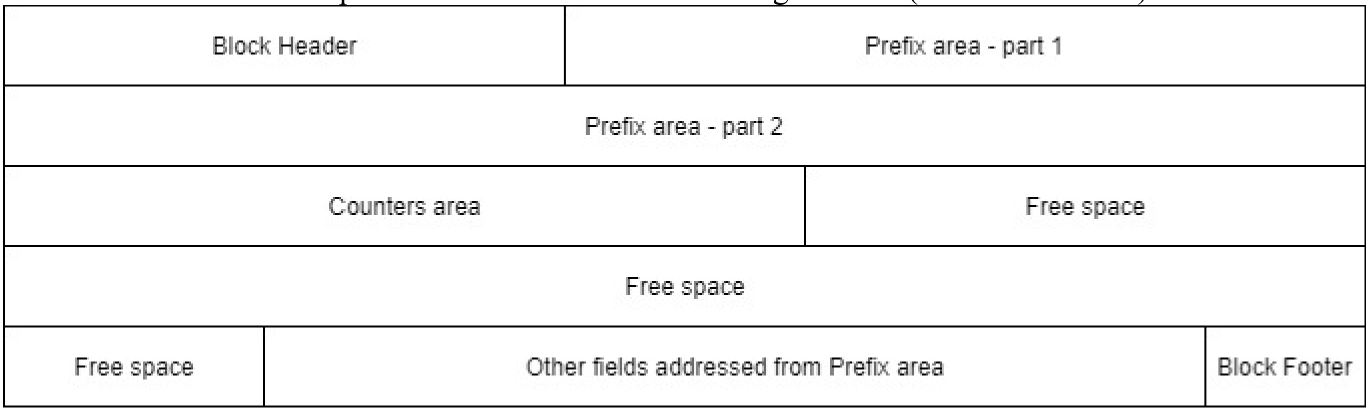
The prefix block (ZVSAMPFX) consists of the first 4096 bytes of every physical file. It contains meta-data defining the file and its attributes. It also contains various counters

The prefix block consists of a block header immediately followed by the prefix area
The prefix block also contains other data fields, these are addressed from the prefix area
The prefix block ends with a block footer. A record pointer list is not present on the prefix block

There are various pointer fields in the prefix area. These point to fields allocated elsewhere in the prefix block. Their exact addresses on the prefix block may vary
The PFXDPAT@, PFXDNAM@, PFXXPAT@, PFXXNAM@ all point to a halfword-prefixed string
PFXDVOL@ and PFXXVOL@ contain foxes (future option)

The Counters area (ZVSAMCTR) directly follows the Prefix area, it is doubleword aligned
This area is expected to move into the catalog dataset in a future release

The overall structure of the prefix block would look something like this (areas not to scale):



Block Header | Prefix area - part 1
Prefix area - part 2
Counters area | Free space
Free space
Free space | Other fields addressed from Prefix area | Block Footer

Prefix Block chain summary

The following table summarizes the way that blocks in the file are chained from the prefix block
The prefix block doesn't reside on any chain

Block Type	Beginning of chain	End of chain
Prefix	foxes	foxes
Spacemap	PFXBMAP	PFXEMAP
Data (in use and free)	PFXBDATA	PFXEDATA
Data (non-first segments)	PFXBSEGM	PFXESEGM
Index (in use and free)	PFXBLVLn	PFXELVLn

Spacemap Blocks

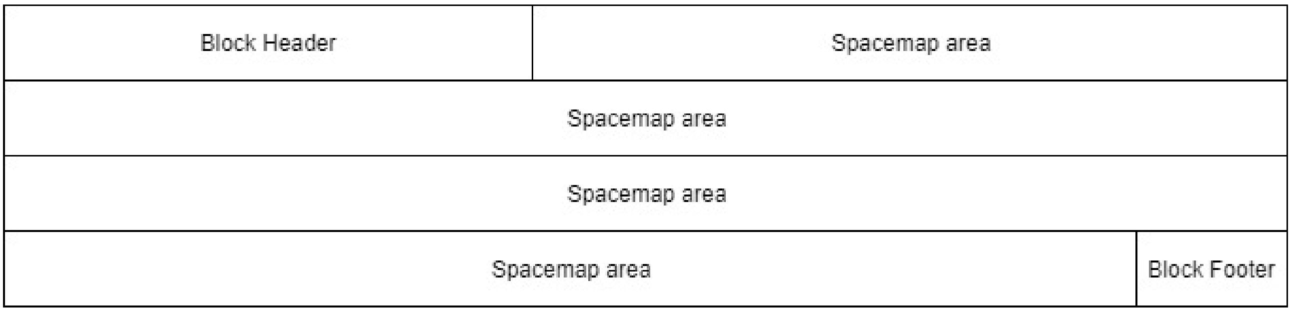
Spacemap blocks (ZVSAMMAP) are used to manage available free space in a component. Each spacemap block has a size that matches the blocksize of all other blocks (except possibly the prefix block) in the component

A component will hold as many spacemap blocks as needed to map all of its allocated blocks, including all spacemap blocks but excluding the prefix block. Whenever a single spacemap block is not enough, the spacemap blocks are chained together by means of the BHDRNEXT/BHDRPREV pointers in the block header area. The spacemap chain starts/ends from the prefix block, fields PFXBMAP/PFXEMAP

When a single spacemap block suffices, PFXBMAP and PFXEMAP will both point to that block

Each spacemap block consists of a block header immediately followed by the spacemap area, which in turn is followed directly by the block footer. No free space exists on a spacemap block. Thus, a spacemap block may indicate blocks that do not exist in the dataset. The bit settings for blocks beyond the PFXHXLRA should all be zero to indicate an unallocated block. zVSAM is aware that any block beyond PFXHXLRA needs to be created and initialized before it can be allocated

Conceptually, the overall structure of a spacemap block would look something like this (areas not to scale):



<table>
<tr><td>Block Header</td><td colspan="2">Spacemap area</td></tr>
<tr><td colspan="3">Spacemap area</td></tr>
<tr><td colspan="3">Spacemap area</td></tr>
<tr><td colspan="2">Spacemap area</td><td>Block Footer</td></tr>
</table>

Record Pointer List Structure (RPTR)

Every block that contains data records contains a record pointer list (ZVSAMRPT). Records are accessible only through their Record Pointer or RPTR. Every entry in the list corresponds with a single record on the block. The last byte of the record's XLRA is the index into the Record Pointer List. Index value of X'00' is reserved for block pointers; values X'01' through X'FF' inclusive are usable as RPTR index values. The difference of 1 always needs to be taken into account when indexing the RPTR list

The RPTR list always follows the block header directly

The number of entries on the RPTR list varies with the number of records stored on the block (BHDR#REC) and is terminated with an entry of foxes to mark the end of the list.

When RPTR_END is set, RPTRREC@ is set to foxes

RPTR_ACT and RPTR_MTY are mutually exclusive. Either one must be set, otherwise the RPTR list is compromised and data access will fail

RPTR_MTY indicates an empty RRDS slot

Segment Prefix (SPX)

All segments begin with a segment prefix (ZVSAMSEG)

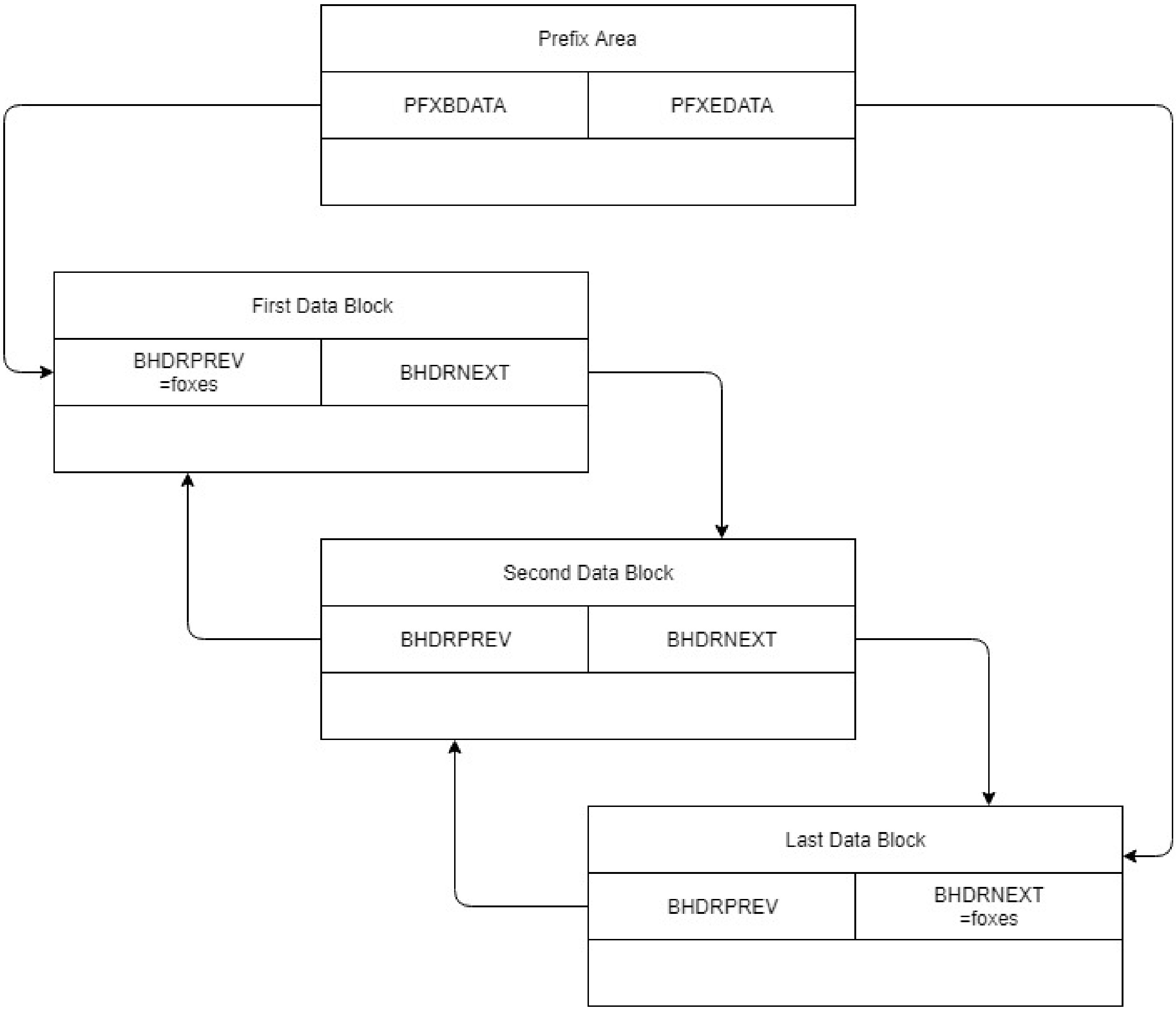
The first segment is on the Data chain and subsequent segments are retrieved via SPXBNEXT

The flag SPXSEGCC indicates the first, middle or last segments

Data Blocks

Data Block Structure (SPANNED=NO)

Assume we have a cluster with three data blocks holding records. The blocks are on the data chain as outlined in the picture below. Please note that all depicted pointers are block pointers. Each pointer thus originates with the indicated field, and ends at the block it points to. The location where the arrows attach has no meaning since it's a block pointer



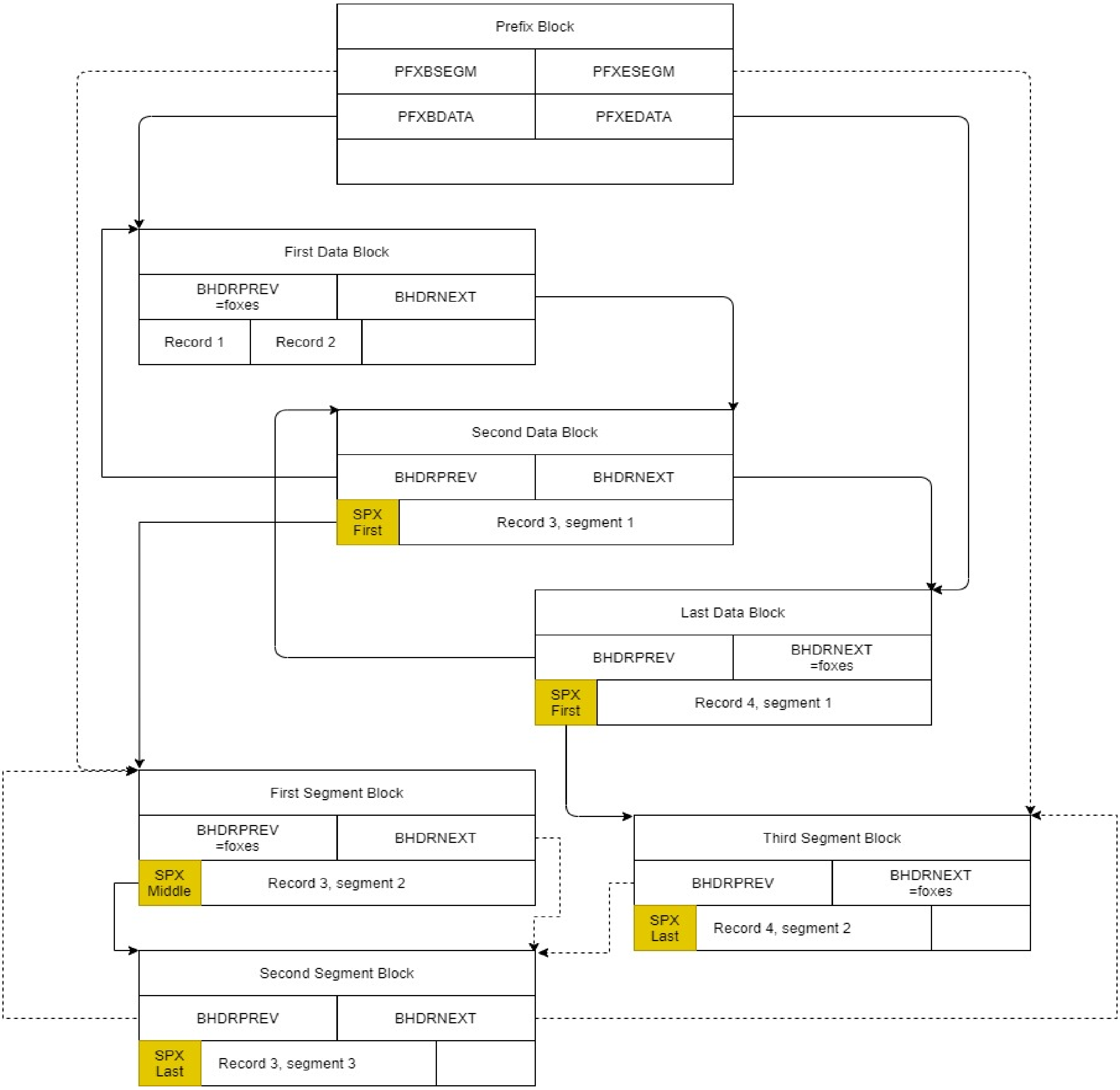
Data Block Structure (SPANNED=YES)

Now suppose we have a cluster with three data blocks, the first block holding two unsegmented records, the second block holding the first segment of a record consisting of three segments and the third block holding the first segment of a record consisting of two segments

In the picture we show the data chain as a solid line (as in the picture above), we show the segment chain as a dotted line, and we show the SPX pointers as a fat line

The picture shows the prefix area's pointer to start/end block of both the data chain and the segment chain

It also shows the first and second block on each chain pointing to one another. Same thing for the second and third block on each chain



All depicted pointers are block pointers

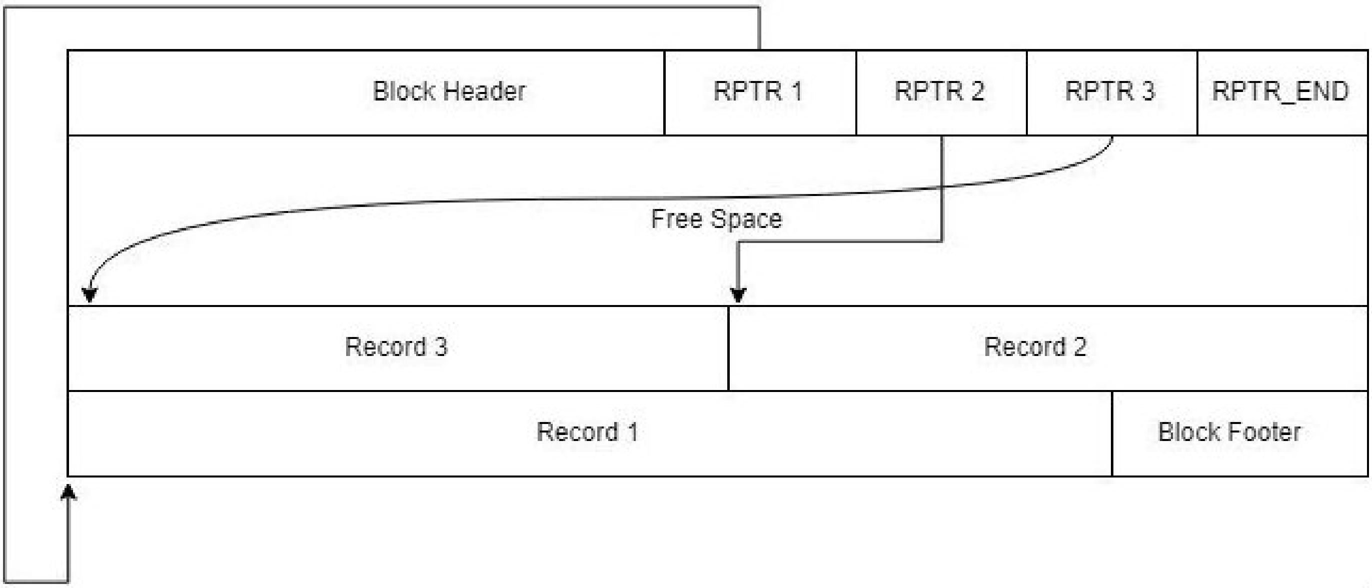
Each pointer originates with the indicated field, and ends at the block it points to

The location where the arrows attach has no meaning since it's a block pointer

Data Block

Each record has an RPTR block, they are created after the Block Header
In addition to the offset, the RPTR contains flags to identify the type and status of each record
RPTR_END marks the end of records in this block

The records are placed in reverse order in the block to consolidate free space at the centre

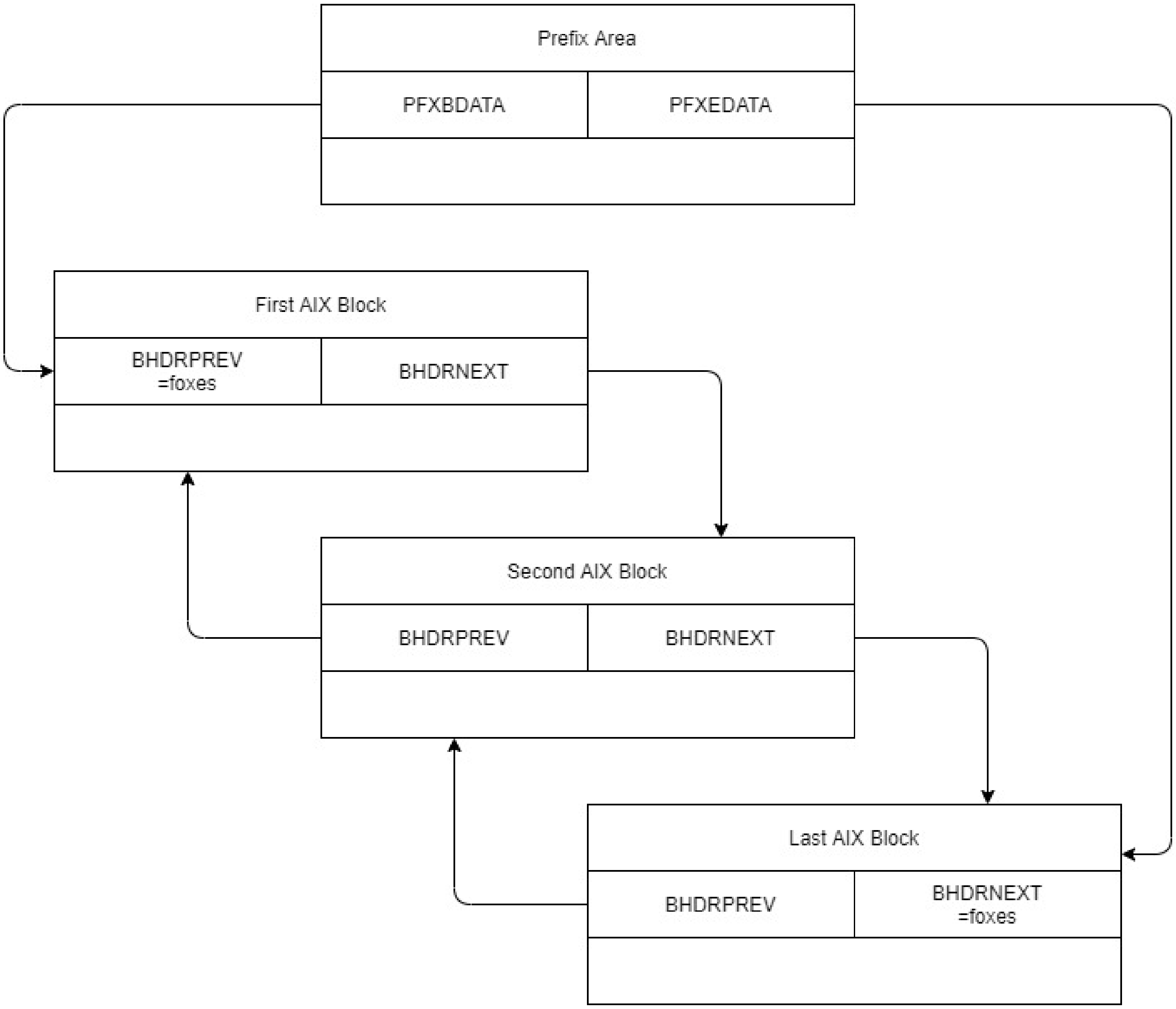


It is possible to reserve an amount of freespace at load time which also applies if a block is split
It is specified in the catalog as DATAFREESPACE=nn, where nn is a percentage of the available space
Only a fixed non-spanned KSDS can specify free space

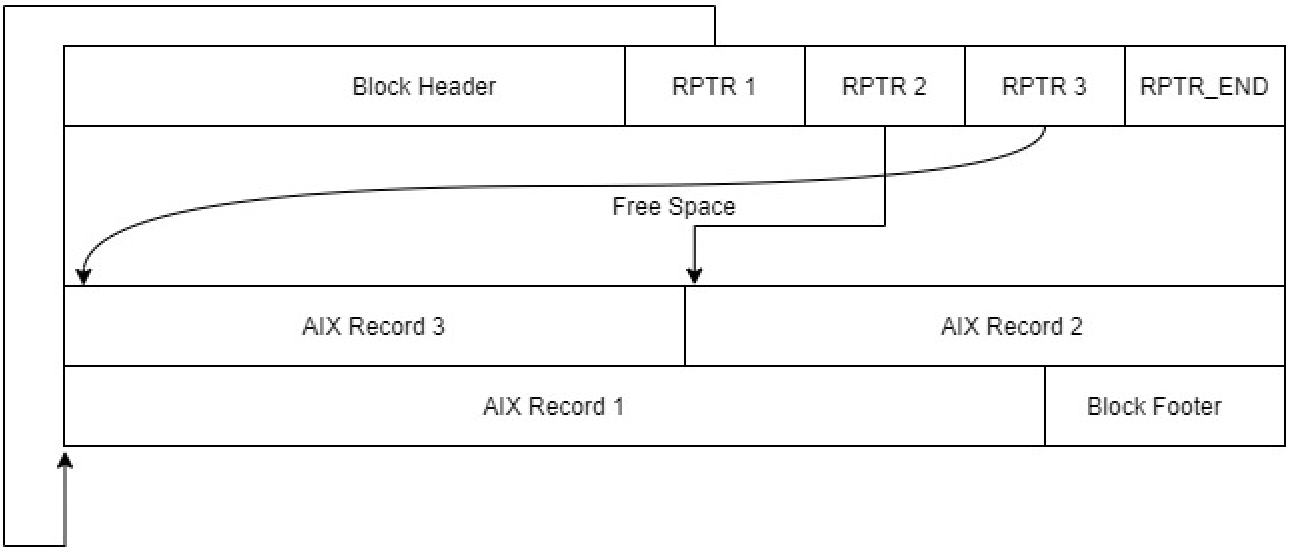
For all types of fixed non-spanned datasets, the available space may not be a multiple of the data record size resulting in unusable space. To correct this use DATAADJUST=YES which will calculate an optimal blocksize less than the specified one

AIX Blocks

AIX Block Structure (Unique)



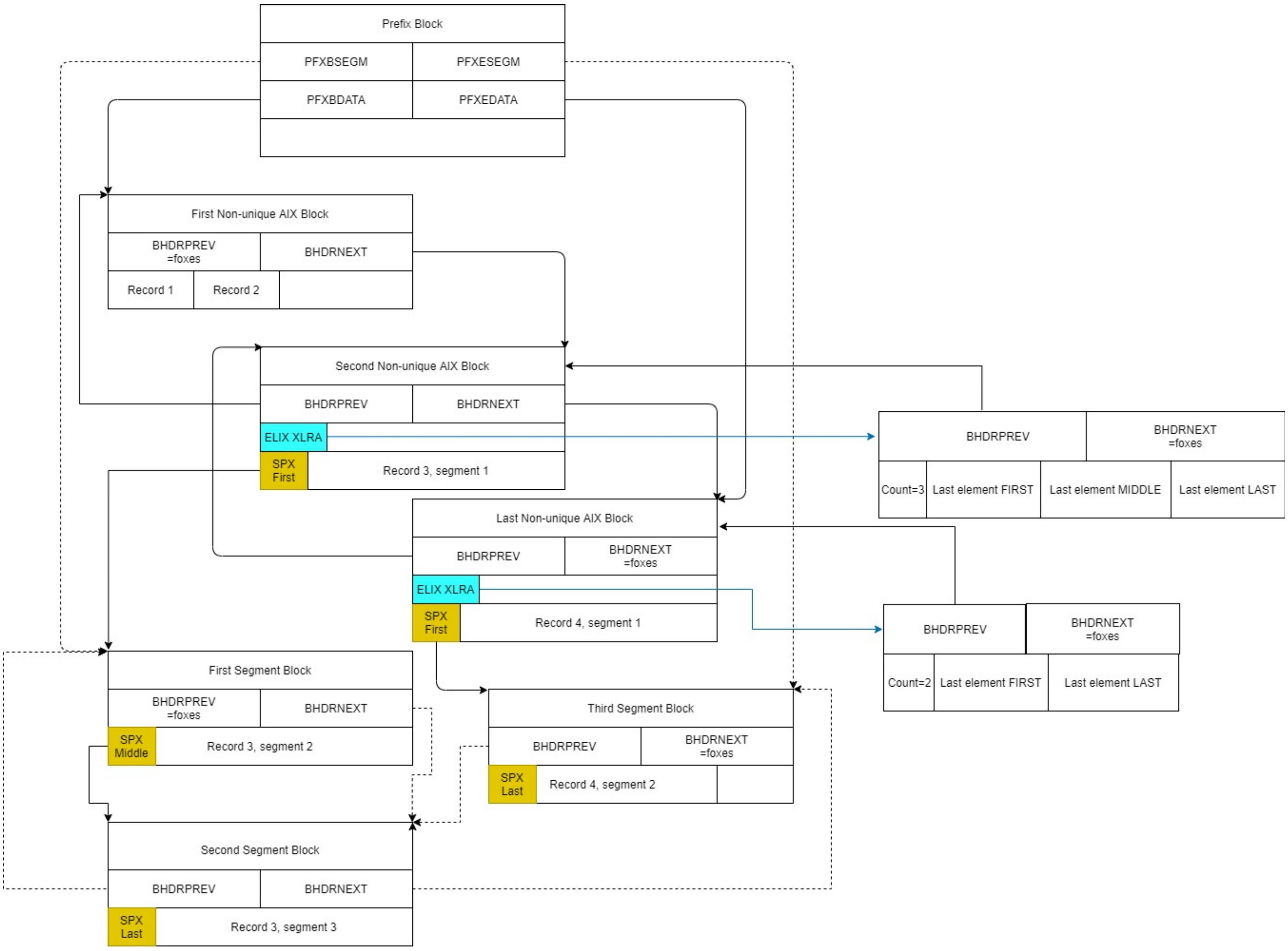
AIX Block (Unique)



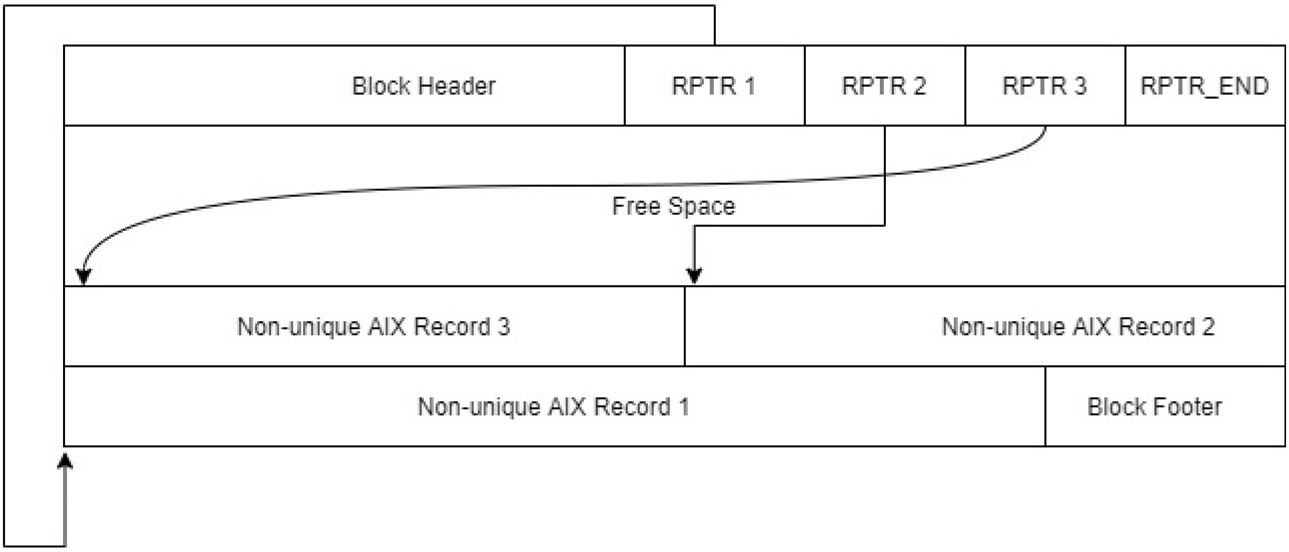
AIX unique records have the following format:

AIX on ...	Record Format
KSDS	AIX key followed by primary key
ESDS	AIX key followed by XRBA(8)

AIX Block Structure (Non-unique)



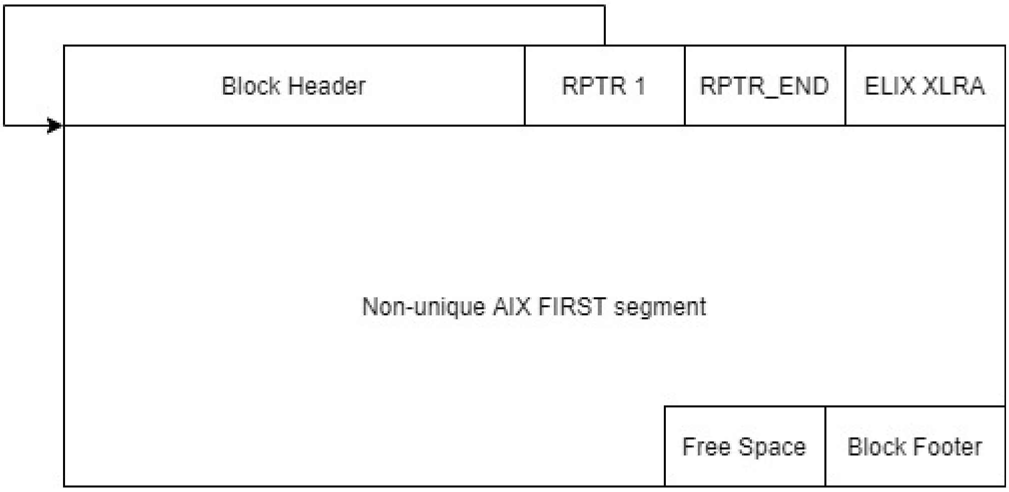
AIX Block (Non-unique and not segmented)



AIX non-unique non-segmented records have the following format:

AIX on ...	Record Format
KSDS	AIX key, an element count n(4) followed by n primary keys
ESDS	AIX key, an element count n(4) followed by n XRBAs(n*8)

AIX Block (Non-unique and segmented)



Each segment contains a whole number of elements
AIX non-unique segmented records have the following formats:

AIX on ...	Record Format of FIRST segment
KSDS	SPX, AIX key, an element count(4) which is the total no. of elements in all segments The actual number of primary keys in this segment can be calculated from SPXSEGLN
ESDS	SPX, AIX key, an element count(4) which is the total no. of elements in all segments The actual number of XLRAs in this segment can be calculated from SPXSEGLN

AIX on ...	Record Format of MIDDLE or LAST segments
KSDS	SPX and a number of primary keys The actual number of primary keys in this segment can be calculated from SPXSEGLN
ESDS	SPX and a number of XLRAs The actual number of XLRAs in this segment can be calculated from SPXSEGLN

ELIX Block

A single ELIX block is created for each non-unique AIX record that is segmented
It has the same blocksize as a Data record

zVSAM lifts the current IBM restriction of 32K elements in a non-unique AIX record, because of this there may be many segments to read to find an element to delete or an insertion point for a new record

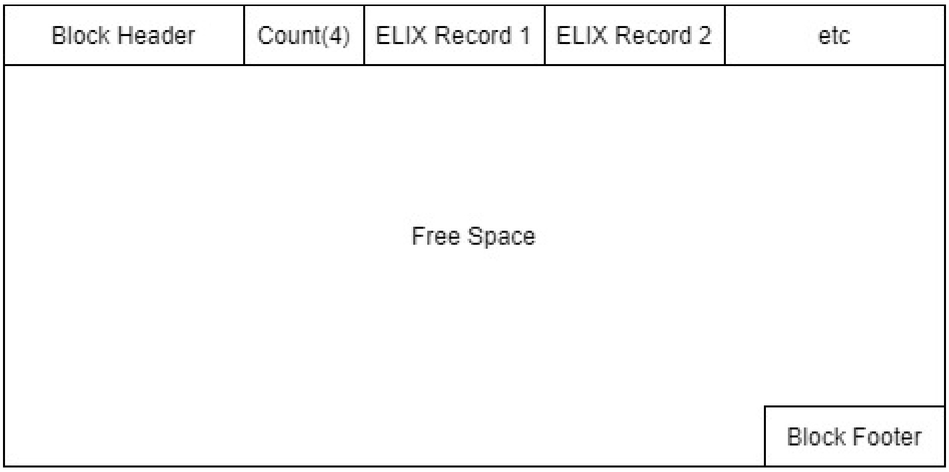
The ELIX Block provides an extra index on the segments and contains the highest element in each segment
As there is currently only one ELIX Block per AIX key this places a limit on the number of elements

When a non-unique AIX is built zREPRO will issue a message on the log like this:

```
   zREPRO AIX MAX ELEMENT LIMIT         87654
```

If the number of elements is too low then rebuild the AIX with a larger blocksize

IBM does not maintain elements in any particular order but for the ELIX structure to work zVSAM will maintain elements in sequence



The ELIX record has the following format:

AIX on ...	Record Format
KSDS	Highest Primary key followed by the XLRA of the segment (always record 1)
ESDS	Highest XRBA followed by the XLRA of the segment (always record 1)

Index Blocks

Each record has an RPTR block, they are created after the Block Header
In addition to the offset, the RPTR contains flags to identify the type and status of each record
RPTR_END marks the end of records in this block
The records are placed in reverse order in the block to consolidate free space at the centre

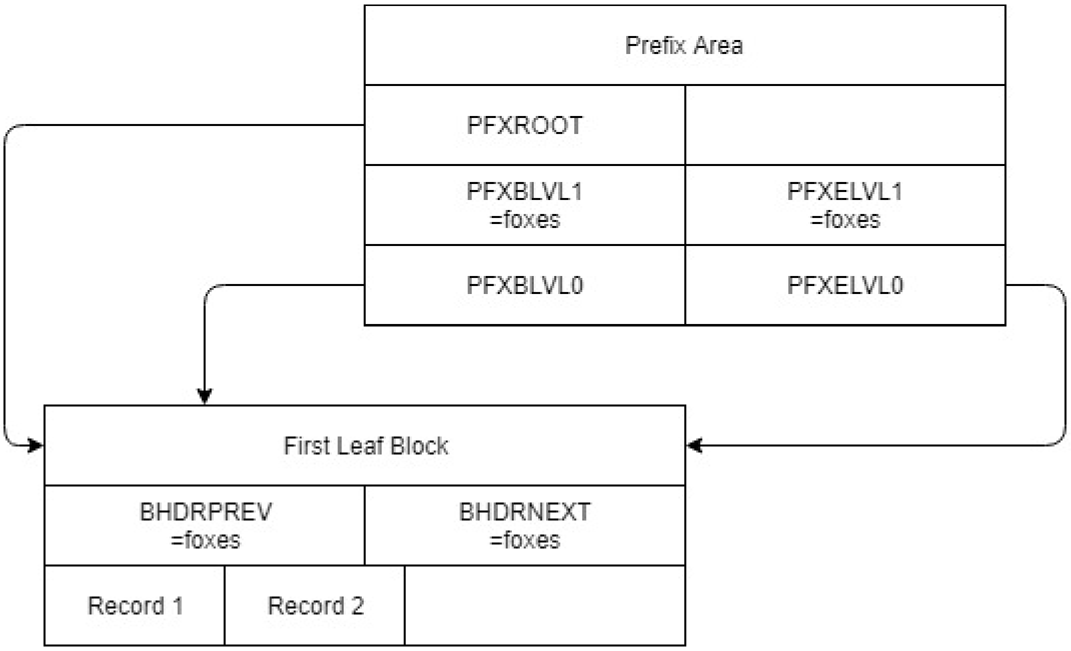
For Level 0 each record is the key (KSDS), XRBA (ESDS) or RRN (RRDS) and is followed by an XLRA
The XLRA is a record pointer to the Data block

For other levels, each record is the highest key, XRBA or RRN followed by an XLRA
The XLRA is a block pointer to the previous level

As each index record is a fixed size it is recommended to specify INDEXADJUST=YES to avoid unusable free space

Index Block Structure: Single level

This example shows an index of only one block, holding two record pointers



Prefix Area

PFXROOT

PFXBLVL1 =foxes

PFXELVL1 =foxes

PFXBLVL0

PFXELVL0

First Leaf Block

BHDRPREV =foxes

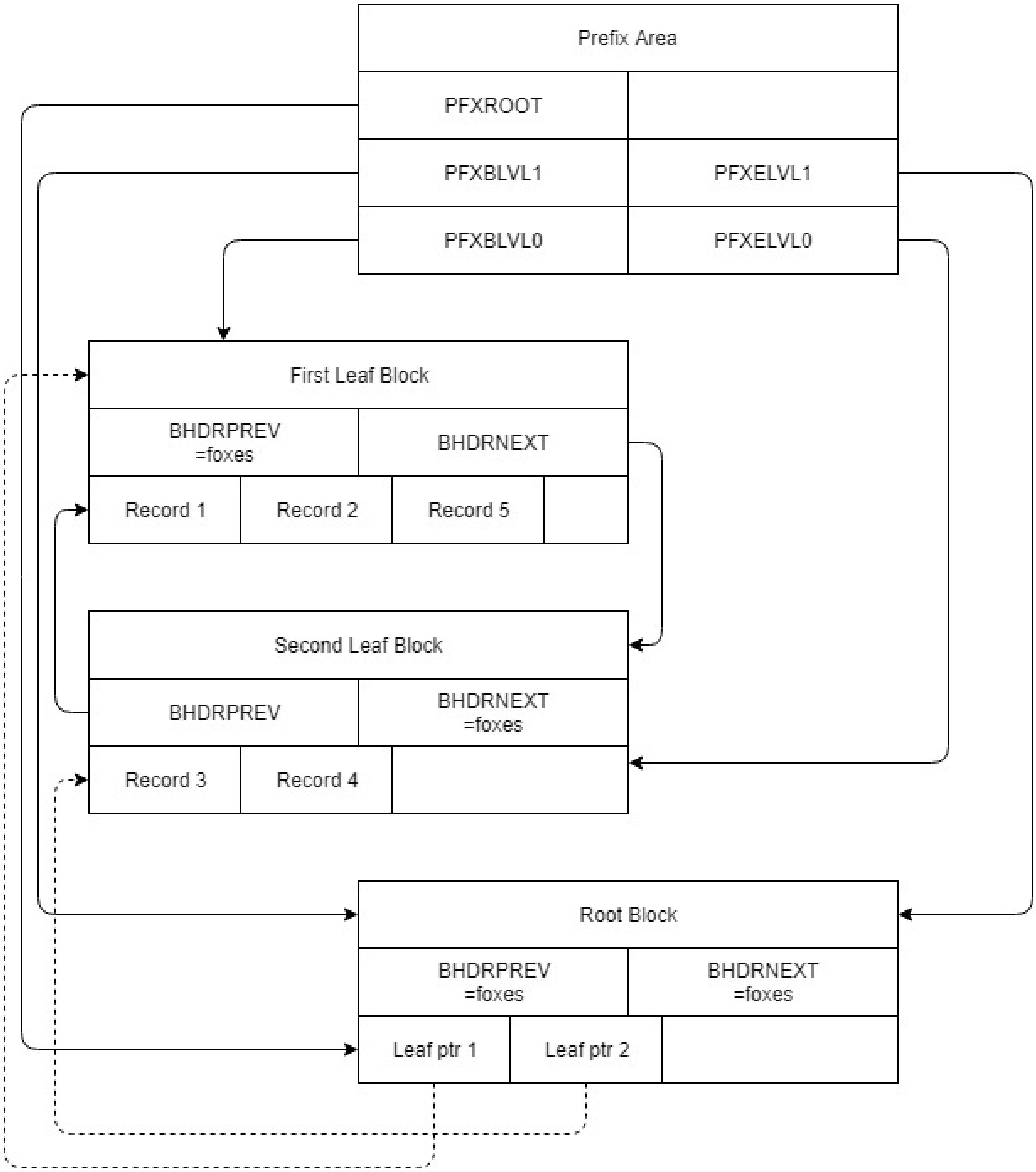
BHDRNEXT =foxes

Record 1

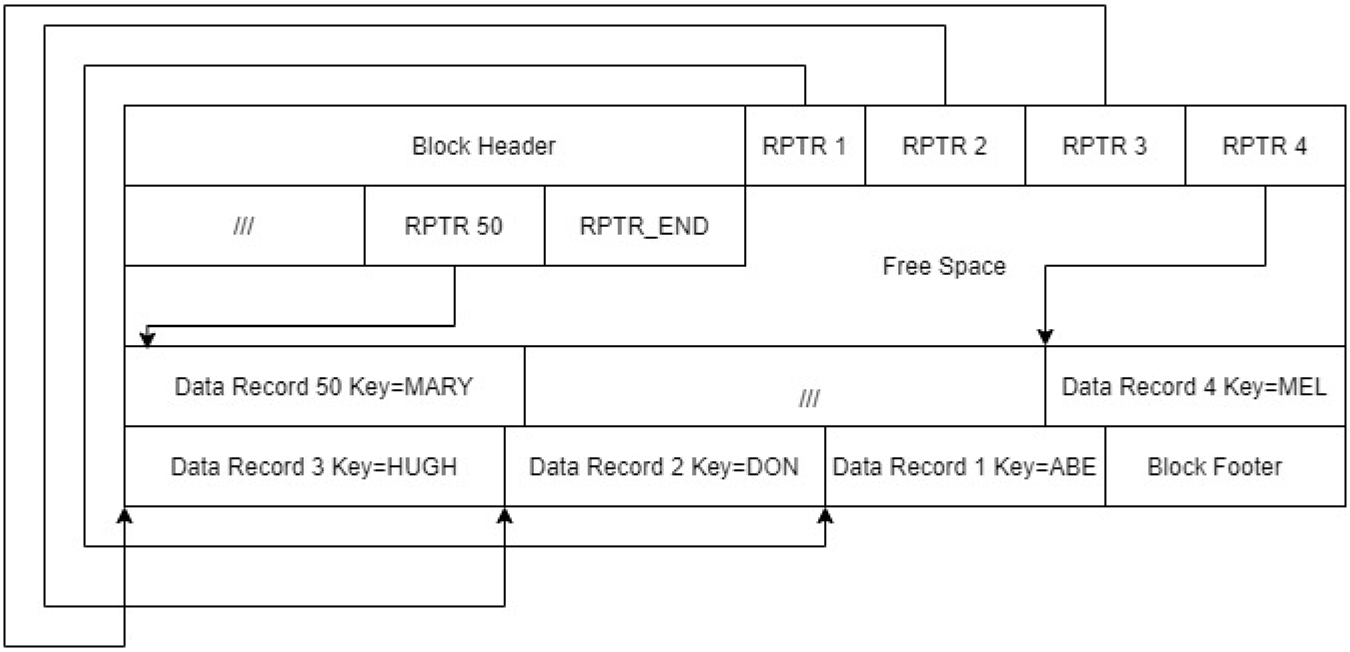
Record 2

Index Block Structure: Two Levels

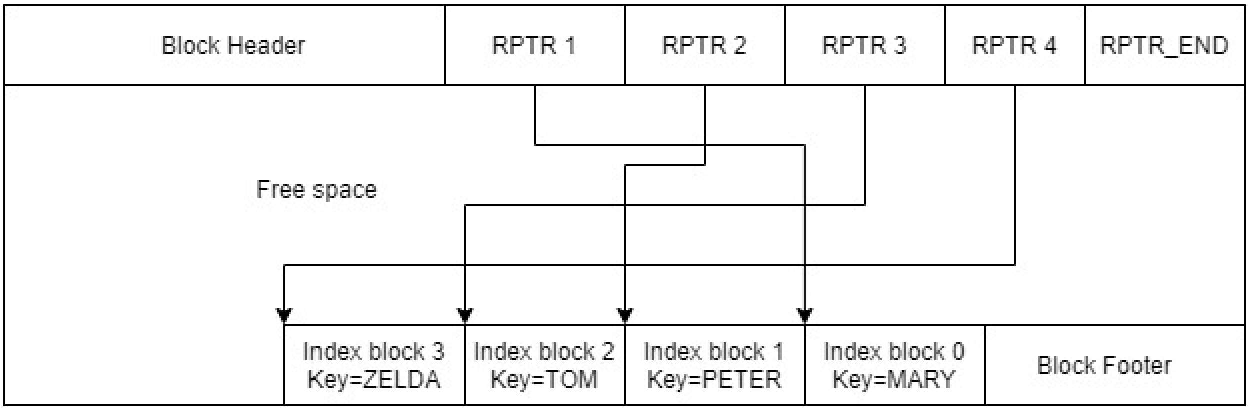
This example shows the index after adding three more record pointers, causing the only index block to overflow and split. Now there are two leaf blocks, still on the LVL0 chain, and a new root block has been created on the LVL1 chain



Index Block Level 0



Index Block other levels



It is possible to reserve an amount of freespace at load time which also applies if a block is split
It is specified in the catalog as INDEXFREESPACE=nn, where nn is a percentage of the available space
Only a fixed non-spanned KSDS can specify free space

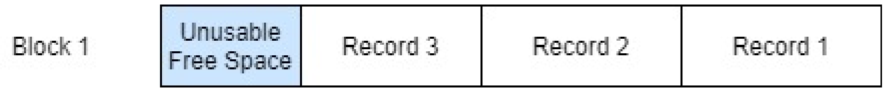
For all types of fixed non-spanned datasets, the available space may not be a multiple of the index record size resulting in unusable space. To correct this use INDEXADJUST=YES which will calculate an optimal blocksize less than the specified one

Structure and Functions by dataset type

KSDS Fixed non-Spanned

F-type records are conceptually stored one after another, filling the block until no space is left.
When the remaining free space is insufficient to accommodate another record, that free space remains unusable. Unusable space can be eliminated by building the dataset with DATAADJUST=YES
Blocks can be allocated with free space for adds (DATAFREESPACE=nn%), when the block is full the block will be split and any new block will have nn% free space

Format:



Block 1	Unusable Free Space	Record 3	Record 2	Record 1

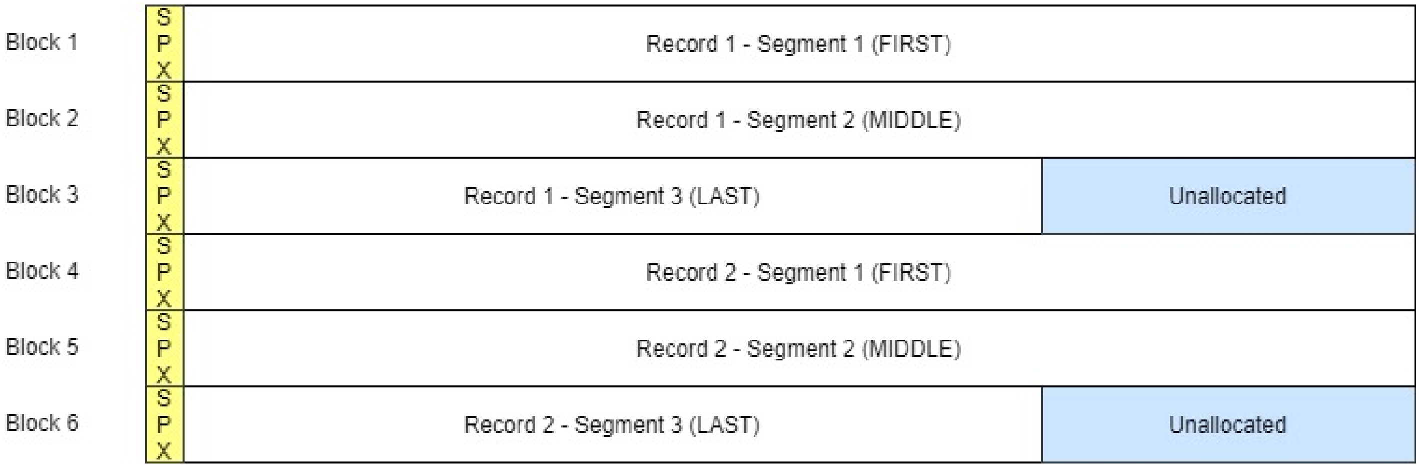
Function	**Notes**
Add	Yes
Update	Yes, the primary key must not be changed
Delete	Yes
Length change	n/a
Access by:	Primary key or AIX key (X)RBA not yet implemented

KSDS Fixed Spanned

FS-type records are conceptually stored one after another, using a block for each segment and starting each record on a new block. Record size is expected to exceed block size, so the record is split into segments, the first segment is created to fill an entire block, and the rest of the record goes into one or more secondary segments which are stored on the next blocks
Each segment is preceded by a Segment Prefix (SPX, marked in yellow)

zVSAM extension: The primary key and any AIX keys need not be in the first segment

Below we show an example where each record requires three segments:


Block 1
SPX
Record 1 - Segment 1 (FIRST)
Block 2
SPX
Record 1 - Segment 2 (MIDDLE)
Block 3
SPX
Record 1 - Segment 3 (LAST)
Unallocated
Block 4
SPX
Record 2 - Segment 1 (FIRST)
Block 5
SPX
Record 2 - Segment 2 (MIDDLE)
Block 6
SPX
Record 2 - Segment 3 (LAST)
Unallocated

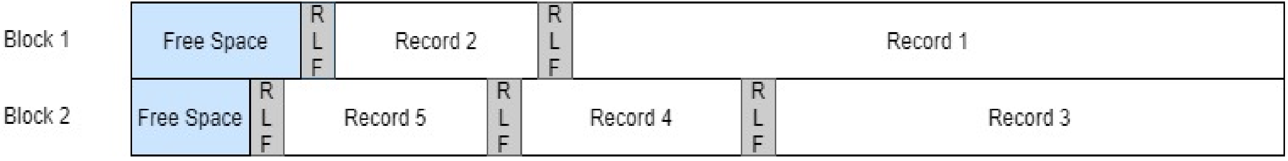
Function	**Notes**
Add	Yes
Update	Yes, the primary key must not be changed
Delete	Yes
Length change	n/a
Access by:	Primary key or AIX key (X)RBA not yet implemented

KSDS Variable non-Spanned

V-type records are conceptually stored one after another, filling the block until no space is left
Every record is preceded by a Record Length Field (RLF, marked in grey)

When remaining free space is insufficient to accommodate another record, that free space remains unallocated (marked in blue) and the record is placed on the next block

Below we show an example showing how various numbers of records might fit into the blocks



Function	**Notes**
Add	Yes
Update	Yes, the primary key must not be changed
Delete	Yes
Length change	Yes When a record is shortened it must not affect the primary key or any AIX key
Access by:	Primary key or AIX key (X)RBA not yet implemented

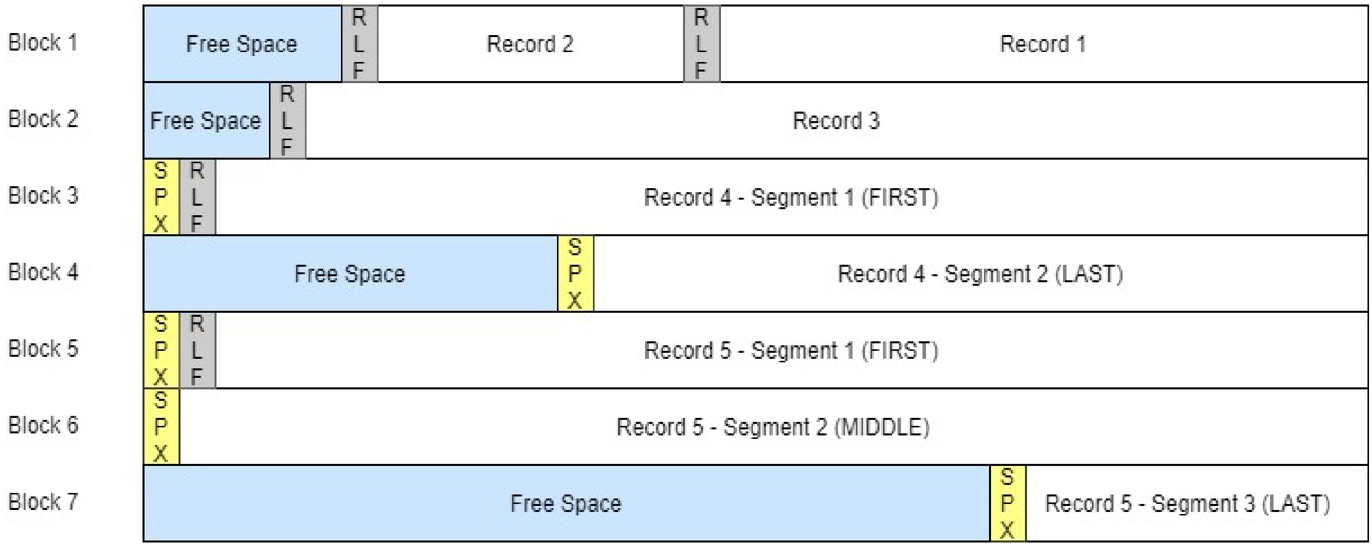
KSDS Variable Spanned

VS-type records are conceptually stored one after another, filling the block until no space is left
Every record is preceded by a Record Length Field (RLF, marked in grey)
When remaining free space is insufficient to accommodate another record, that free space remains unallocated (marked in blue) and the record is placed on the next block

Only if the record size exceeds the usable block size is the record is split into segments and each segment is prefixed with a Segment Prefix. The first segment is created to fill an entire block, and the rest of the record goes into one or more secondary segments which are stored on the next blocks
Each segment is preceded by a Segment Prefix (SPX, marked in yellow)

zVSAM extension: The primary key and any AIX keys need not be in the first segment

Below we show an example showing how various numbers of records might fit into the blocks of the file, or how a single record might occupy multiple blocks of the file

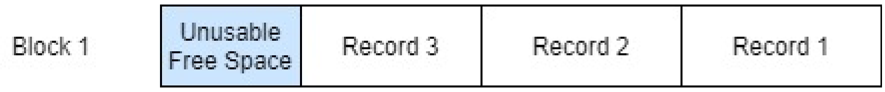


Function	**Notes**
Add	Yes
Update	Yes, the primary key must not be changed
Delete	Yes
Length change	Yes When a record is shortened it must not affect the primary key or any AIX key
Access by:	Primary key or AIX key (X)RBA not yet implemented

ESDS Fixed non-Spanned

F-type records are conceptually stored one after another, filling the block until no space is left. When the remaining free space is insufficient to accommodate another record, that free space remains unusable. Unusable space can be eliminated by building the dataset with DATAADJUST=YES

Format:



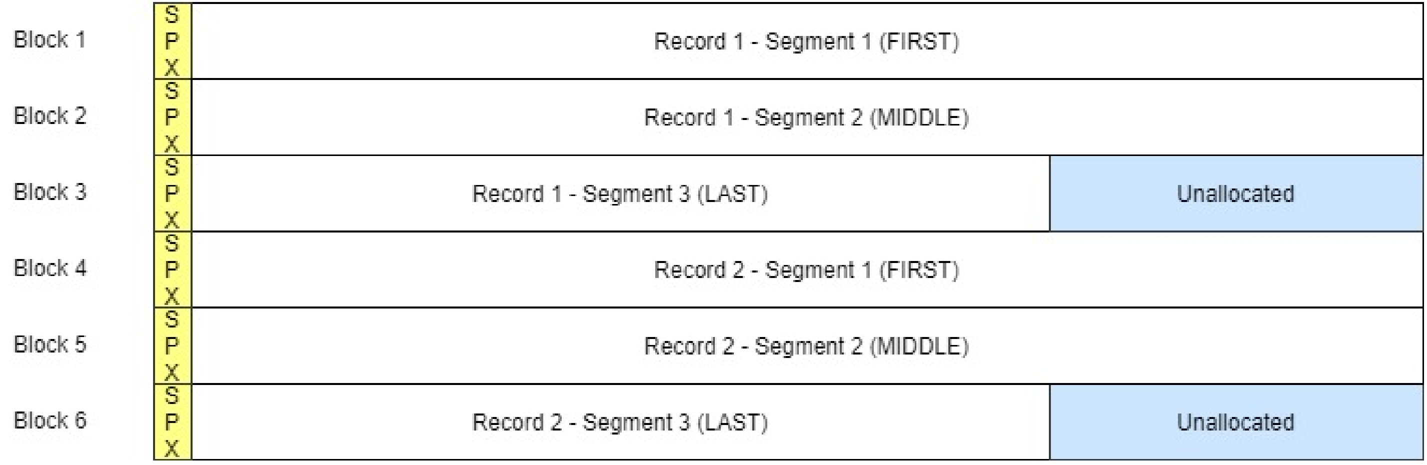
Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	No
Length change	n/a
Access by:	(X)RBA or AIX key

ESDS Fixed Spanned

FS-type records are conceptually stored one after another, using a block for each segment and starting each record on a new block. Record size is expected to exceed block size, so the record is split into segments, the first segment is created to fill an entire block, and the rest of the record goes into one or more secondary segments which are stored on the next blocks
Each segment is preceded by a Segment Prefix (SPX, marked in yellow)

zVSAM extension: Any AIX keys need not be in the first segment

Below we show an example where each record requires three segments:



Block 1 | SPX | Record 1 - Segment 1 (FIRST)

Block 2 | SPX | Record 1 - Segment 2 (MIDDLE)

Block 3 | SPX | Record 1 - Segment 3 (LAST) | Unallocated

Block 4 | SPX | Record 2 - Segment 1 (FIRST)

Block 5 | SPX | Record 2 - Segment 2 (MIDDLE)

Block 6 | SPX | Record 2 - Segment 3 (LAST) | Unallocated

Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	No
Length change	n/a
Access by:	(X)RBA or AIX key

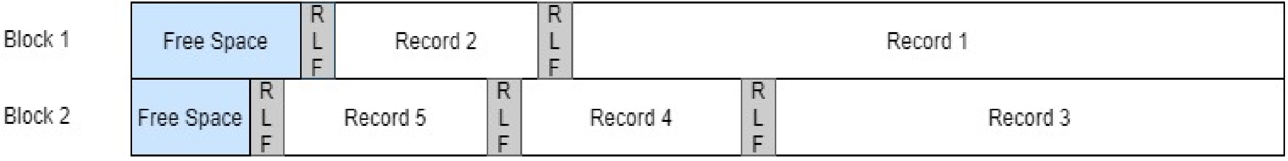
ESDS Variable non-Spanned

V-type records are conceptually stored one after another, filling the block until no space is left
Every record is preceded by a Record Length Field (RLF, marked in grey)

When remaining free space is insufficient to accommodate another record, that free space remains unallocated (marked in blue) and the record is placed on the next block

This dataset type is a zVSAM extension

Below we show an example showing how various numbers of records might fit into the blocks



Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	No
Length change	No
Access by:	(X)RBA or AIX key

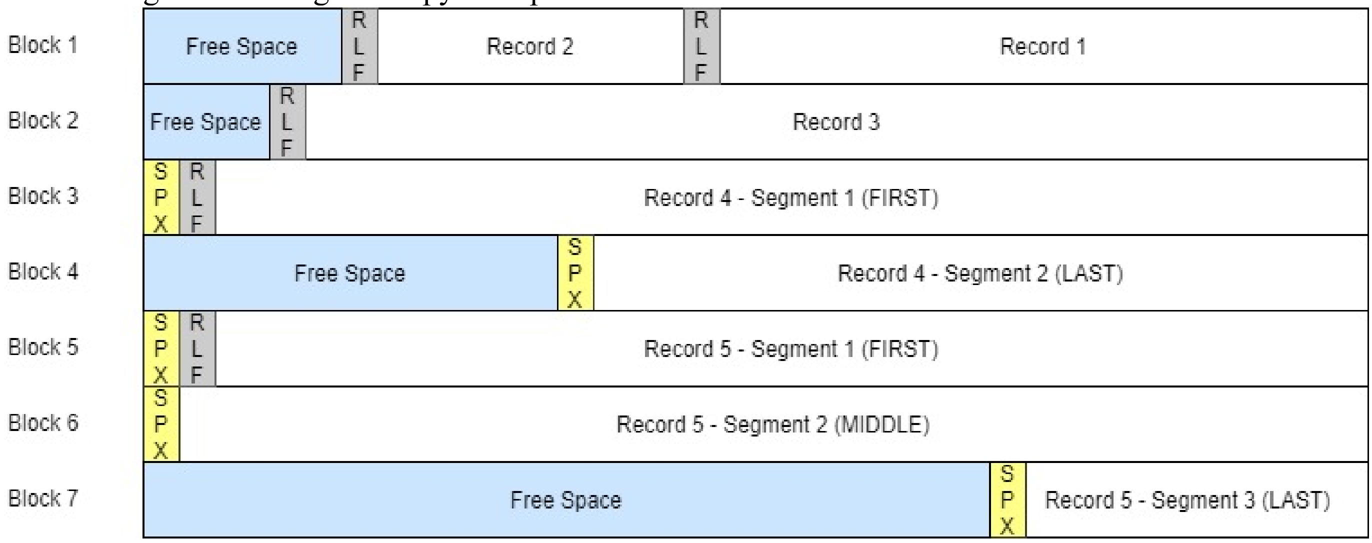
ESDS Variable Spanned

VS-type records are conceptually stored one after another, filling the block until no space is left
Every record is preceded by a Record Length Field (RLF, marked in grey)
When remaining free space is insufficient to accommodate another record, that free space remains unallocated (marked in blue) and the record is placed on the next block

Only if the record size exceeds the usable block size is the record is split into segments and each segment is prefixed with a Segment Prefix. The first segment is created to fill an entire block, and the rest of the record goes into one or more secondary segments which are stored on the next blocks
Each segment is preceded by a Segment Prefix (SPX, marked in yellow)

This dataset type is a zVSAM extension
zVSAM extension: Any AIX keys need not be in the first segment

Below we show an example showing how various numbers of records might fit into the blocks of the file, or how a single record might occupy multiple blocks of the file

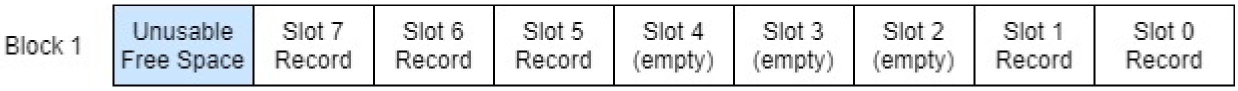


Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	No
Length change	No
Access by:	(X)RBA or AIX key

RRDS Fixed non-Spanned

F-type records are conceptually stored one after another, filling the block until no space is left. When the remaining free space is insufficient to accommodate another record, that free space remains unusable. Unusable space can be eliminated by building the dataset with DATAADJUST=YES

An RRDS consists of slots (RRNs) which may or may not contain a record
Empty slots are initially binary zeros with RPTR_MTY set



Block 1	Unusable Free Space	Slot 7 Record	Slot 6 Record	Slot 5 Record	Slot 4 (empty)	Slot 3 (empty)	Slot 2 (empty)	Slot 1 Record	Slot 0 Record

Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	Yes, slots may not be deleted. RPTR_MTY is set
Length change	n/a
Access by:	RRN

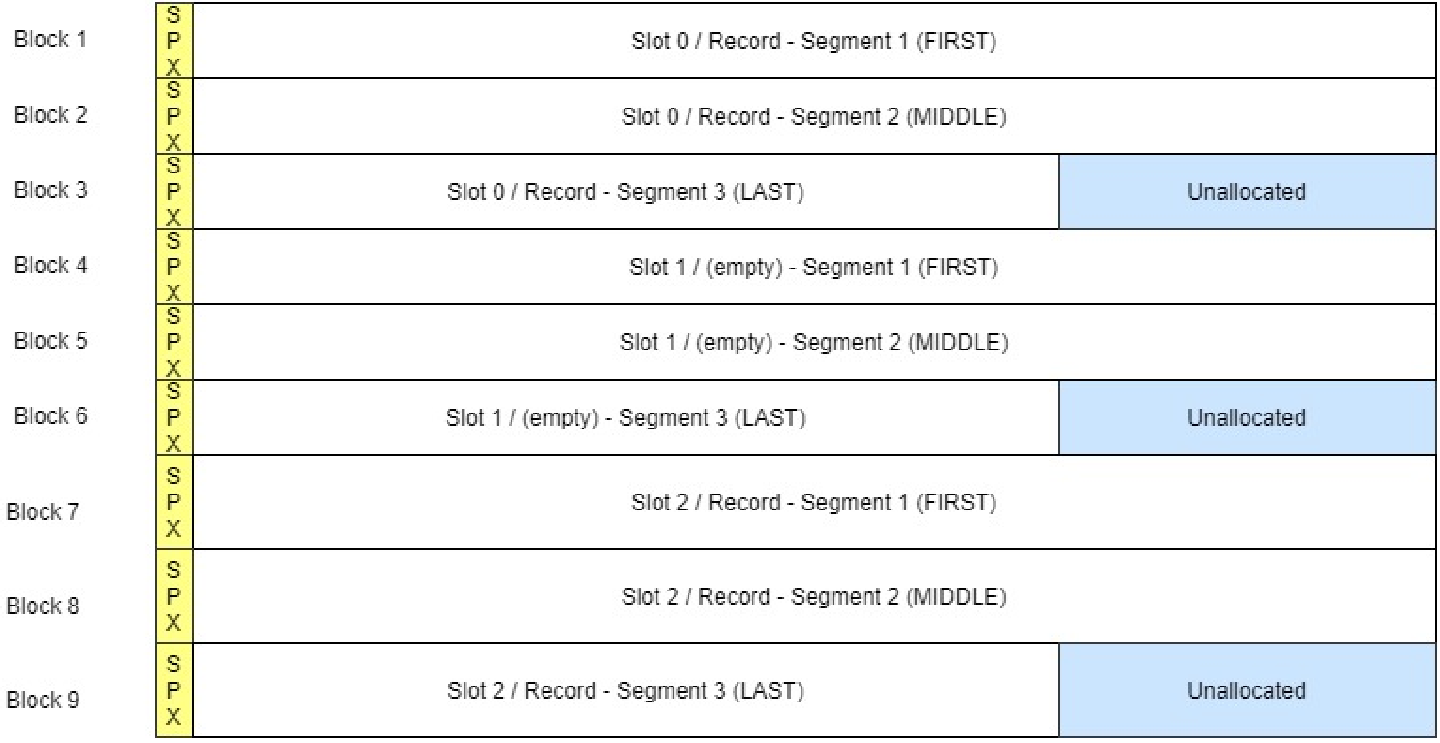
RRDS Fixed Spanned

FS-type records are conceptually stored one after another, using a block for each segment and starting each record on a new block. Record size is expected to exceed block size, so the record is split into segments, the first segment is created to fill an entire block, and the rest of the record goes into one or more secondary segments which are stored on the next blocks
Each segment is preceded by a Segment Prefix (SPX, marked in yellow)

An RRDS consists of slots (RRNs) which may or may not contain a record
Empty slots are initially binary zeros with RPTR_MTY set

This dataset type is a zVSAM extension

Below we show an example where each record requires three segments:


Block 1
SPX
Slot 0 / Record - Segment 1 (FIRST)
Block 2
SPX
Slot 0 / Record - Segment 2 (MIDDLE)
Block 3
SPX
Slot 0 / Record - Segment 3 (LAST)
Unallocated
Block 4
SPX
Slot 1 / (empty) - Segment 1 (FIRST)
Block 5
SPX
Slot 1 / (empty) - Segment 2 (MIDDLE)
Block 6
SPX
Slot 1 / (empty) - Segment 3 (LAST)
Unallocated
Block 7
SPX
Slot 2 / Record - Segment 1 (FIRST)
Block 8
SPX
Slot 2 / Record - Segment 2 (MIDDLE)
Block 9
SPX
Slot 2 / Record - Segment 3 (LAST)
Unallocated

Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	Yes, slots may not be deleted. RPTR_MTY is set
Length change	n/a
Access by:	RRN

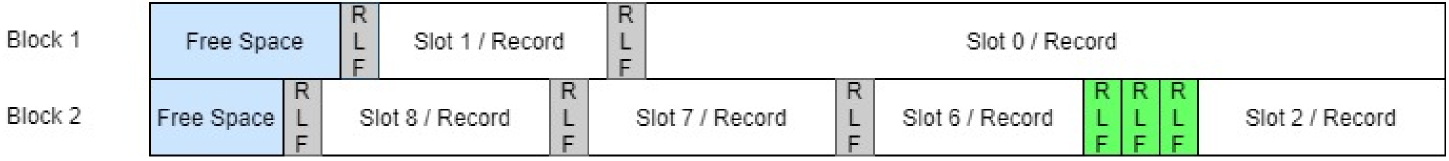
RRDS Variable non-Spanned

V-type records are conceptually stored one after another, filling the block until no space is left
Every record is preceded by a Record Length Field (RLF)

An RRDS consists of slots (RRNs) which may or may not contain a record
Empty slots consist of a dummy RLF containing X'00000004' with RPTR_MTY set, these are shown in green in the diagram. Non-empty slots have a grey RLF

When remaining free space is insufficient to accommodate another record, that free space remains unallocated (marked in blue) and the record is placed on the next block

Below we show an example showing how various numbers of records might fit into the blocks



Function	Notes
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	Yes, slots may not be deleted. RPTR_MTY is set The record is replaced by a dummy RLF and the space is reclaimed
Length change	Yes
Access by:	RRN

RRDS Variable Spanned

VS-type records are conceptually stored one after another, filling the block until no space is left
Every record is preceded by a Record Length Field (RLF)

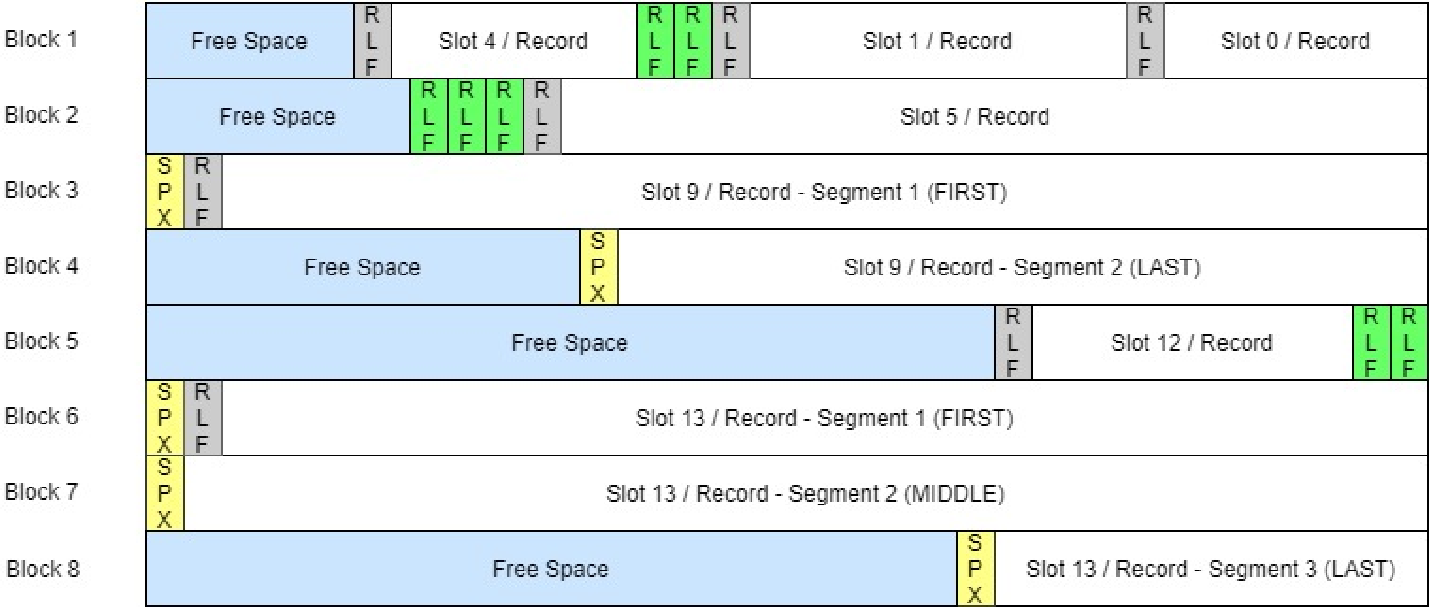
An RRDS consists of slots (RRNs) which may or may not contain a record
Empty slots consist of a dummy RLF containing X'00000004' with RPTR_MTY set, these are shown in green in the diagram. Non-empty slots have a grey RLF

When remaining free space is insufficient to accommodate another record, that free space remains unallocated (marked in blue) and the record is placed on the next block

When a record length exceeds the available space in a block the record is split into segments, the first segment is created to fill an entire block, and the rest of the record goes into one or more secondary segments which are stored on the next blocks
Each segment is preceded by a Segment Prefix (SPX, marked in yellow)

This dataset type is a zVSAM extension

Below we show an example showing how various numbers of records might fit into the blocks



Function	**Notes**
Add	Yes, but only to the end of the dataset
Update	Yes
Delete	Yes, slots may not be deleted. RPTR_MTY is set The record is replaced by a dummy RLF and the space is reclaimed For segmented records, the freed blocks are marked as available
Length change	Yes
Access by:	RRN

Logical processes for RPL-based requests

POINT function

GET function

Prefix counter field CTRNEXCP needs to be incremented whenever a block is needed that does not yet reside in a buffer. If buffers need to be written out to make room for a block that needs to be read, then the CTRNEXCP counter needs to be incremented as well.

Prefix counter field CTRNRETR needs to be incremented for every block accessed, whether it needs to be read, or already resides in a buffer does not matter for this count field

If buffers need to be written out in order to allocate a buffer for a block that needs to be read, then prefix counter CTRNNUIW needs to be incremented

PUT function

Prefix counter field CTRAVSPAC must be updated. When consuming free space to allocate a new record, reduce CTRAVSPAC with space consumed. When allocating a new block (or splitting an existing block) add the blocksize – (block header, block footer, RPTR area) and reduce with amount used up (record length, including RDW/SPX and RPTR). When lengthening a record, subtract the difference; when shortening a record, add the difference

Prefix counter fields CTRHALCRBA and CTRENDRBA should be updated whenever a record is added beyond the current value of these fields. CTRENDRBA also should be updated in case the last record in the component is lengthened

Prefix counter field CTRNCIS should be incremented whenever a block needs to be split

Prefix counter field CTRNEXCP needs to be incremented whenever a block needs to be written out. This occurs when the cluster was opened with MACRF=NDF. With MACRF=DFR no writes are forced and no EXCP needs to be counted

Prefix counter fields CTRNINSR and CTRNLOGR need to be incremented whenever a new record is added to the component

Prefix counter field CTRNRETR needs to be incremented for every block accessed, whether it needs to be read, or already resides in a buffer does not matter for this count field. For basic put operation this is irrelevant, but for updating an index or an AIX reads may be needed and should be counted for the component being read

When an existing record is updated, then CTRNUPDR needs to be incremented

When adding a record the record length (including SPX/RDW, but excluding RPTR) needs to be added to the prefix counter field CTRSDTASZ. When a record is updated to a different length, then the difference in length needs to accounted into the CTRSDTASZ field.

When a user write is forced then the prefix counter field CTRNUIW needs to be incremented. This happens when a put is issued to a cluster that was opened with MACRF=NDF. For clusters opened with MACRF=DFR writing is done by zVSAM when the buffer is needed for a different block. These writes are counted in the CTRNNUIW field

The prefix counter field CTRAVGRL contains the value of CTRSDTASZ / CTRNLOGR rounded up to the nearest integer. The field should be updated whenever either or both of the input values are changed

The prefix counter field CTRLOKEY@ is the offset to a length and a value of PFXKYLEN bytes

The value should be updated whenever a record is inserted that has a lower key than the current lowest key.

ERASE function

Prefix counter field CTRNDELR should be incremented for every successful erase operation

Prefix counter field CTRNEXCP needs to be incremented whenever a block is needed that does not yet reside in a buffer. If buffers need to be written out to make room for a block that needs to be read, then the CTRNEXCP counter needs to be incremented as well

Prefix counter field CTRNLOGR needs to be decremented for every successful erase operation

Prefix counter field CTRNRETR needs to be incremented for every block accessed, whether it needs to be read, or already resides in a buffer does not matter for this count field. For basic erase operation this is irrelevant, but for updating an index or an AIX reads may be needed and should be counted for the component being read

When erasing a record the record length (including SPX/RDW, but excluding RPTR) needs to be subtracted from the prefix counter field CTRSDTASZ

When a user write is forced then the prefix counter field CTRNUIW needs to be incremented. This happens when an erase is issued to a cluster that was opened with MACRF=NDF. For clusters opened with MACRF=DFR writing is done by zVSAM when the buffer is needed for a different block. These writes are counted in the CTRNNUIW field

The prefix counter field CTRAVGRL contains the value of CTRSDTASZ / CTRNLOGR rounded up to nearest integer. The field should be updated after every successful erase operation

The prefix counter field CTRLOKEY@ is the offset to a length and value of PFXKEYLN bytes

The value should be updated whenever the record is erased with the current lowest key

Prefix counter field CTRNEXCP needs to be incremented whenever a block is needed that does not yet reside in a buffer

CHECK function

ENDREQ function

VERIFY function

Locking

Addenda

API for ACB-based interfaces

TESTCB ACB macro parameters

All supported parameters are implemented compatibly with IBM's VSAM implementation.
For details, please refer to the relevant IBM manual.

For ease of access a short summary follows here:
ACB=addr required to indicate the ACB to be tested

All other keywords function the same way that they do on a SHOWCB ACB request. Please see the preceding chapter for details here

MF=I or omitted
Specifies the standard form of the TESTCB to generate an inline CBMR and an inline call to the CBMR handler.

MF=L Specifies the list form of the TESTCB macro which generates an inline CBMR but no call to the CBMR handler.

MF=(L,addr) Specifies the list form of the TESTCB macro to generate a remote CBMR at the indicated location. No call to the CBMR handler is generated.

MF=(L,addr,label)
Same as MF=(L,addr) but label will be equated to the length of the CBMR.

MF=(E,addr) Specifies the execute form of the TESTCB macro to generate code that will dynamically modify the CBMR at the indicated address according to the parameters specified before calling the CBMR handler.

MF=(G,addr) Specifies the generate form of the TESTCB macro to generates code to modify the indicated CBMR as specified by the other parameters and to call the CBMR handler.

MF=(G,addr,label)
Same as MF=(G,addr) but label will be equated to the length of the CBMR

TESTCB EXLST macro parameters

All supported parameters are implemented compatibly with IBM's VSAM implementation.
For details, please refer to the relevant IBM manual.

For ease of access a short summary follows here:
EXLST=addr required to indicate the EXLST to be tested

mod modifier, can optionally be specified after each routine address.
Values: A or N for Active or Not-active.
When this modifier is specified, only Equal or Not-Equal condition can be returned.
The secondary modifier of L (for Load from Linklib) is not supported.

MF=I or omitted
Specifies the standard form of the TESTCB to generate an inline CBMR and an inline call to the CBMR handler.

MF=L Specifies the list form of the TESTCB macro which generates an inline CBMR but no call to the CBMR handler.

MF=(L,addr) Specifies the list form of the TESTCB macro to generate a remote CBMR at the indicated location. No call to the CBMR handler is generated.

MF=(L,addr,label)
Same as MF=(L,addr) but label will be equated to the length of the CBMR.

MF=(E,addr) Specifies the execute form of the TESTCB macro to generate code that will dynamically modify the CBMR at the indicated address according to the parameters specified before calling the CBMR handler.

MF=(G,addr) Specifies the generate form of the TESTCB macro to generates code to modify the indicated CBMR as specified by the other parameters and to call the CBMR handler.

MF=(G,addr,label)
Same as MF=(G,addr) but label will be equated to the length of the CBMR

API for RPL-based interfaces

TESTCB RPL macro parameters

All supported parameters are implemented compatibly with IBM's VSAM implementation.
For details, please refer to the relevant IBM manual.

For ease of access a short summary follows here:

RPL=addr required to indicate the RPL to be tested

FTNCD=nr Values used for FTNCD and their meaning can be found in the IBM manual "DFSMS Macro Instructions for Datasets", chapter "Return and Reason Codes", section "Component Codes"

RBA=nr – zVSAM supports this keyword only for ESDS. For any other type of cluster a value of foxes will be assumed by default.

MF=I or omitted
Specifies the standard form of the TESTCB to generate an inline CBMR and an inline call to the CBMR handler.

MF=L Specifies the list form of the TESTCB macro which generates an inline CBMR but no call to the CBMR handler.

MF=(L,addr) Specifies the list form of the TESTCB macro to generate a remote CBMR at the indicated location. No call to the CBMR handler is generated.

MF=(L,addr,label)
Same as MF=(L,addr) but label will be equated to the length of the CBMR.

MF=(E,addr) Specifies the execute form of the TESTCB macro to generate code that will dynamically modify the CBMR at the indicated address according to the parameters specified before calling the CBMR handler.

MF=(G,addr) Specifies the generate form of the TESTCB macro to generates code to modify the indicated CBMR as specified by the other parameters and to call the CBMR handler.

MF=(G,addr,label)
Same as MF=(G,addr) but label will be equated to the length of the CBMR

POINT macro parameters

GET macro parameters

PUT macro parameters

ERASE macro parameters

CHECK macro parameters

ENDREQ macro parameters

VERIFY macro parameters

List of changes

Date	Author	Description
2018-09-16	Abe Kornelis	Remove SPX from VS records that have only a single segment. Change order and numbering of chapters Move macro parameter descriptions to addendum Expand chapter on compatibility In the addendum for GENCB ACB add explanation on MF usage
2018-09-18	Abe Kornelis	Various small changes as suggested by Hugh Sweeney Moved zACB and zEXLST layout paragraphs to the addenda.
2018-09-20	Abe Kornelis	Various small changes as suggested by Melvyn. See mail dated 2018-09-19 22:19
2018-09-27	Abe Kornelis	Added content for MODCB ACB, including addendum.
2018-09-29	Abe Kornelis	Added content for SHOWCB ACB, including addendum
2018-10-01	Abe Kornelis	Added content for TESTCB ACB, including addendum
2018-10-07	Abe Kornelis	Added comment on CBMR layout to chapters on GENCB ACB, MODCB ACB, SHOWCB ACB and TESTCB ACB. Parm AM=VSAM added to GENCB ACB chapter. Added content for GENCB EXLST, including addendum
2018-10-08	Abe Kornelis	Added content for MODCB EXLST, including addendum Added content for SHOWCB EXLST, including addendum Added content for TESTCB EXLST, including addendum
2018-10-09	Abe Kornelis	CBMR split into header and separate tail sections CBMR header description added
2018-10-10	Abe Kornelis	Minor changes as suggested by Melvyn's mail dd 2018-10-09 23:40 Addition of chapter titles for RPL-based interfaces to addenda.
2018-10-11	Abe Kornelis	Added CBMR description – body for ACB
2018-10-13	Abe Kornelis	Added CBMR description – body for EXLST Added RPL macro description, including addendum Added GENCB RPL macro description, including addendum Added MODCB RPL macro description, including addendum Added SHOWCB RPL macro description, including addendum Added TESTCB RPL macro description, including addendum
2018-10-15	Abe Kornelis	Added ACBPFX pointer to zACB layout
2018-10-16	Abe Kornelis	Added CBMR description – body for RPL ACBTYPE → ACBSTYPE Removed ACBMACR3_NLW and ACBMACR3_MODE ACBCUEL → ACBUEL ACBOCK → ACBLOCK
2018-10-21	Abe Kornelis	ACB ADR/KEY improved keyword description in addendum ACB IN/OUT improved keyword description in addendum ACB DDNAME improved keyword description in ACB macro chapter and the addendum Unsupported parameters and keywords on ACB, EXLST, RPL changed from “flagged as error” to “ignored”
2018-10-22	Abe Kornelis	Add description of prefix block, including counters area.

Date	Author	Description
		Updated addendum for SHOWCB/TESTCB with reference to source of data for each keyword. Added prefix field PFXIXLVL. Added instructions for RPL-based operations on how to maintain prefix counter fields. Added description of spacemap block.
2018-10-23	Abe Kornelis	Specify that SHOWCB/TESTCB for RBA/XRBA is supported for ESDS only. Foxes for any other component.
2018-10-24	Abe Kornelis	Add description for block header, block, footer, record pointer list
2018-10-26	Abe Kornelis	Added description of open macro logic
2018-10-27	Abe Kornelis	Added eyecatcher to the prefix area, moved record length and key info fields to beginning of prefix area BHDRPREV/NEXT on prefix block documented as being foxes
2018-10-28	Abe Kornelis	Added ACBVER to zACB layout Added Area to Terminology chapter Max. block size reduced from 2G to 16MB Added ACBXPFX to zACB layout Added description of open execution logic
2018-11-21	Abe Kornelis	In API on macro interfaces improved wording for handling of (as yet) unsupported macro parameters. Add ATRB=VESDS for TESTCB ACB Improved picture and text on spacemap block layout
2018-11-22	Abe Kornelis	BHDRPREV/NEXT details expanded Removed PFXBSEG/PFXESEG
2018-11-25	Abe Kornelis	Spacemap area structure. Segmented records were missing. Added.
2018-11-29	Abe Kornelis	Added diagrams to chapter on block header structure.
2018-12-02	Abe Kornelis	Added alternative diagram for chaining segments. Preferred solution not yet determined And added drawings for chaining index blocks
2018-12-09	Abe Kornelis	Structure of Physical files: ELIX added to the list of block types Block Header Structure: BHDRFLGS changed to BHDRFLG1 and added BHDRFLG2 with BHDR_ELX
2018-12-17	Abe Kornelis	Put PFXBSEG/PFXESEG back in Corrected typos in drawings for explaining BHDRNEXT/PREV RPTR_END no longer all foxes, foxes only for RPTRREC@ Added 4 date fields to the prefix structure for creation and last update timestamps for both data and index component. MF=omitted changed to MF= in various locations
2019-01-06	Abe Kornelis	Added various fields to RPL

Date	Author	Description
2019-01-23	Melvyn Maltz	Version 2.0 Took over the D&L, not documenting spelling, syntax, bad references and other trivia Added hyperlinks Amended CBMRACB fields Amended CBMRRPL fields Removed RPL TIMEOUT...VTAM only Updated Prefix Block DSECT
2019-02-17	Melvyn Maltz	Version 2.1 Corrected diagram "Segmented Data Block Structure" CBMR RPL – Added GEN MOD SHOW TEST column CBMR ACB – Added GEN MOD SHOW TEST column CBMRACB RMODE31 – Added description
2019-02-21	Melvyn Maltz	Corrections to RPL DSECT. Added RPLFEEDB Note added to AIXPC
2019-02-22	Melvyn Maltz	Corrections and updates to ACB DSECT
2019-02-25	Melvyn Maltz	Added note to MODCB ACB about turning off MACRF=OUT
2019-02-26	Melvyn Maltz	Revised the OPEN Execution Logic table to distinguish V1 from V2
2019-03-02	Melvyn Maltz	Revised the EXLST DSECT
2019-03-03	Melvyn Maltz	Added section EXLST Macro Logic CBMR EXLST – Added GEN MOD SHOW TEST column Added _MODS in preparation for MODCB
2019-03-10	Melvyn Maltz	Removed sections on Logical Processes that don't involve calls to Java eg. MODCB. These are already well described
2019-03-12	Melvyn Maltz	Added Return and Reason Codes to MODCB ACB, RPL and EXLST
2019-03-17	Melvyn Maltz	Revised all sections on OPEN and CLOSE Added "CLOSE Macro Logic" Removed OPCL DSECT, replaced with list formats
2019-03-18	Melvyn Maltz	Added Data Block diagram Most references and diagrams for LDS removed, too long term
2019-04-15	Melvyn Maltz	Added hyperlinks to EXLST CBMR Modifiers
2019-04-17	Melvyn Maltz	Added UPAD= and RLSWAIT= Although not supported, they need to exist Amended the EXLST CBMR keyword values to fit them in Added WAREA, LENGTH and LOC to EXLST CBMR
2019-05-19	Melvyn Maltz	Added WAREA, LENGTH and LOC to ACB CBMR Amended the ACB CBMR keyword values to fit them in
2019-06-26	Melvyn Maltz	Added WAREA, LENGTH and LOC to RPL CBMR Added Return and Reason Codes to GENCB ACB, RPL and EXLST
2019-06-29	Melvyn Maltz	Changed PFXMAPOF from 3 to 4 bytes, adjusted following offsets
2019-06-30	Melvyn Maltz	Version 2.2 Open Execution Logic, removed references to OC24/OC31/OCPL
2019-07-06	Melvyn Maltz	RPLACB changed to RPLDACB to match IBM

Date	Author	Description
2019-07-07	Melvyn Maltz	CBMR EXLST: Added CBMRXL_AREA and _EXLST Marked fields used for SHOWCB
2019-07-08	Melvyn Maltz	zVSAM V2 compatibility with zVSAM V1 Added item 4 about re-assembling modules with OPEN
2019-08-17	Melvyn Maltz	EXLST Macro corrected Extra comments added to clarify missing parms for GENCB
2019-08-24	Melvyn Maltz	AM=VSAM added to Macros SHOWCB EXLST, UPAD and RLSWAIT removed, IBM doesn't support them. JRNAD will return zero
2019-10-09	Melvyn Maltz	Added subfields to RPLFEEDB Added RPLCXRBA
2019-10-13	Melvyn Maltz	Added CBMRACB_ACB and CBMRRPL_RPL for SHOWCB Renamed CBMRRPL_CRBA to CBMRRPL_RBA
2019-10-20	Melvyn Maltz	Renamed CBMRRPL_AREA to CBMRRPL_RECAREA and added CBMRRPL_AREA to resolve conflict between SHOWCB RPL AREA= and FIELDS=AREA Renumbered CBMRRPL_RPLLEN to X'6E' to avoid conflict with CBMRXL_XLSTLEN Added CBMRRPL_TRANSID for SHOWCB
2019-10-27	Melvyn Maltz	Corrected syntax to all forms of MF=L SHOWCB, ACB, EXLST and RPL have all 3 lengths: ACBLEN, EXLLEN and RPLLEN New section added “No block type specified” and SHOWCB for extracting lengths only New section added “CBMR description-body for no block specified”
2019-11-13	Melvyn Maltz	SHOWCB ACB FIELDS revised Added several X~ fields as SHOWCB counters expect 4 bytes and the CTR fields are 8 bytes
2020-01-26	Melvyn Maltz	SHOWCB ACB revisions CBMRACB_RMODE31 and CBMRACB_SHRPL codes changed to avoid conflict with CBMRRPL_RPLLEN and CBMRXL_XLSTLEN
2020-02-24	Melvyn Maltz	CBMRACB_SDTASZ added as an 8-byte field CTRSDTA renamed to CTRSDTASZ References to a value of zero returned removed
2020-03-01	Melvyn Maltz	Corrections made to the description of CTRLOKEY@ This is an offset to LL+key
2020-03-02	Melvyn Maltz	Added CBMRACB_AREA
2020-03-15	Melvyn Maltz	Added CBMRACB_XNINSR

Date	**Author**	**Description**
2020-04-28	Melvyn Maltz	Reconstructed the Macro description sections to be more like that of the VSAM Macro manual Separate chapter for the xCB Macro MF= with hyperlinks to it from each of them, duplicate MF= descriptions removed
2020-05-28	Melvyn Maltz	Added diagrams for Data and Index (L0 and Ln) blocks
2020-05-29	Melvyn Maltz	Version 2.3 Removing individual sections on Fixed, Variable etc. and ESDS, KSDS and RRDS Removing Concepts Replaced with all 12 dataset type diagrams with descriptions Removed Displaced Record structure Added SPX format Added AIX structures and formats Added ELIX structures and formats Removed Counters area values, these are referenced in the macros themselves Added Implied OPEN table Added Close execution logic (incomplete) All OPEN-related, EXLST/Exit-related and CLOSE-related doc now in single chapters Addenda items moved to the appropriate chapter or deleted Deleted all DSECTs, too difficult to maintain both the DSECT and the doc The DSECTs are fully commented and offsets can be seen in an assembly CBMR details moved to the xCB chapter and simplified