# **VSAM File I/O for REXX Procedures**

## **PURPOSE**

The RXVSAMIO external REXX function allows a REXX procedure to access a VSAM dataset of either a 'KSDS', 'ESDS', 'PATH', 'RRDS', 'VRDS', 'SAME', 'SAMI', or 'AIX' dataset type. Passed information is a required DLBL name and the access request to be performed. Some access requests have additional, optional parameters with implied defaults. This routine cannot be used to access NOCIFORMAT files.

The passed DLBL name must also refer to a pre-initialized REXX stem variable of the same name. The elements of that stem variable provide additional information for this external REXX function and this external REXX function will add or update elements of that stem variable as needed to provide additional information to the caller. Some of the other things that stem variable will then contain, on request, is returned dataset attributes and statistics.

## **FEATURES**

Why would you use this function? Why not use the provided VSAMIO function or obtain the Source Forge project RXVSAM as your method for accessing VSAM files from a REXX procedure? The issue is that both of those methods, unlike a higher-level language such as COBOL, force your REXX procedure to do *most* of the work and *only* provide VSAM I/O services. Meaning: Your REXX code must do all of the work of parsing and converting the returned VSAM record data, for subsequent use by your REXX procedure, and then have to reconvert and concatenate the data in order to send it back to the VSAM file for update, insert, or output processing.

Conversely, this function provides all of the following features:

* VSAM keyed, RBA, RRN, and sequential I/O access;
* Forward and backward next record retrieval;
* All input, update, and output methods;
* Copybook-like layout support for multi-field keys and records;
* Field data is always stored in separate REXX variables for your convenience;
* Automatic parsing and conversion to standard REXX data formats;
* Automatic reconversion and concatenation back to standard VSAM data formats; and,
* Inquiry for most of the IDCAMS LISTCAT-like dataset attributes and statistics.

NOTE: There is only one access string and one record layout per file. So, if more than one position in the same file is needed or different record layouts are needed for a multi-format file, then separate DLBL statements are required in order to handle either of  
these situations. Up to nine file definitions at a time are accepted by this function.

## **USAGE**

The information passed on the function call has the following general syntax:

Call RXVSAMIO ddname, request <, request\_arg2 <, request\_arg3 > >;

And the following are all of the possible request formats (note that the request arguments may be spelled out, of course, but only as few letters for uniqueness are actually checked):

Call RXVSAMIO ddname, 'ATTRS';  
 Where: no OPEN or CLOSE required for attributes & statistics but,  
 if already open, the dataset is automatically closed.

Call RXVSAMIO ddname, 'OPEN' <, 'INPUT'|'UPDATE'|'OUTPUT'|'APPEND'|'RESET'  
 <, 'SEQUENCE'|'RANDOM'|'DYNAMIC' > >;  
 Where: 'INPUT' and 'SEQUENCE' are the defaults,  
 'RESET' is the same as 'OUTPUT' but for REUSE, and  
 'DYNAMIC' enables skip-sequential processing.

Call RXVSAMIO ddname, {'FWDPOS'|'BWDPOS'} <, 'KEY' <, 'GTEQ'|'EQUAL' > |'RBA'>;  
 Where: 'KEY' and 'GTEQ' are the defaults , and  
 this performs VSAM POINT function processing—only  
 setting the file position and direction (i.e., no record read,  
 but 'GTEQ' is ignored for backward positioning).

Call RXVSAMIO ddname, 'READ' <, 'NEXT'|'PREV'|'KEY' <, 'EQUAL'|'GTEQ' > |'RBA'>;  
 Where: 'NEXT' is the default (or,  
 if 'KEY' is specified, then 'EQUAL' is the default)  
 and, if opened for UPDATE, always causes a CI lock.

Call RXVSAMIO ddname, 'WRITE';  
 Where: on write, automatically releases any prior locks.

Call RXVSAMIO ddname, 'UPDATE';  
 Where: a prior successful READ is required, first.

Call RXVSAMIO ddname, 'DELETE';  
 Where: a prior successful READ is required, first.

Call RXVSAMIO ddname, 'CLOSE';

Where: { } encloses required request arguments;  
 < > encloses optional request arguments;  
 | separates mutually exclusive options; and,  
 , commas are required to separate arguments.

Otherwise, the stem variable mentioned is where you provide the data definitions (see INPUT, below) that allow this function to parse the VSAM records, after retrieval, so that the caller does not have to do this themselves. The parsed (and converted) field data is also returned in the stem variable—a separate entry for each field definition given. At WRITE or UPDATE time, this function also uses the provided data definitions to rebuild the complete VSAM record from the individual field data found in the stem variable.

The caller must provide two sets of data definitions in the stem variable—one set for the key layout and one set for the record layout. This is documented under INPUT, below. These data definitions provide information for up to seven different field data types—as follows (note that these data type codes may be spelled out in the data definitions, of course, but only as few letters for uniqueness are actually checked):

### Table 1: Data Types

| **Data type code** | **Description** |
| --- | --- |
| CHAR | is string data up to 256 bytes in length that does not require conversion. For a key field, of course, this is limited to 255 bytes.  The COBOL equivalent is a PIC A or PIC X field and USAGE IS DISPLAY. |
| HEX | is string data up to 128 bytes in length that is automatically converted to 2‑byte (per character) hexadecimal format when returned to the caller and automatically converted back to 1-byte character format in the VSAM record.  The COBOL equivalent is a PIC X field and USAGE IS DISPLAY. |
| BINARY | is a signed, binary integer 2-, 4-, or 8-bytes in length that is automatically converted to REXX format when returned to the caller and automatically converted back to signed, binary format in the VSAM record:  2-bytes is up to 32,767 max,  4-bytes is up to 2,147,483,647 max, and  8-bytes is up to 9,223,372,036,854,775,807 max.  The COBOL equivalent is a PIC S9 field without a V position and USAGE IS COMP or USAGE IS BINARY. |
| UBIN | is an unsigned, binary integer 1-, 2-, or 4-bytes in length that is automatically converted to REXX format when returned to the caller and automatically converted back to unsigned, binary format in the VSAM record:  1-byte is up to 255 max,  2-bytes is up to 65,535 max, and  4-bytes is up to 4,294,967,295 max.  There is no COBOL equivalent for this because COBOL's idea of no sign is to force the sign to be positive. But, e.g., VSAM RBAs are 4-byte, unsigned binary and VSAM record lengths are 2-byte, unsigned binary. |
| PACKED | is a signed, packed-decimal number up to 16 bytes long (31 digits plus a sign) that is automatically converted to REXX format when returned to the caller and automatically converted back to signed, packed-decimal format in the VSAM record.  The COBOL equivalent is a PIC S9 field with or without a V position and USAGE IS COMP-3 or USAGE IS PACKED-DECIMAL. |
| ZONED | is a signed, zoned-decimal number up to 31 bytes long that is automatically converted to REXX format when returned to the caller and automatically converted back to signed, zoned-decimal format in the VSAM record.  The COBOL equivalent is a PIC S9 field with or without a V position and USAGE IS DISPLAY. |
| IZONED | is an unsigned, zoned-decimal integer up to 31 bytes long that is automatically converted to REXX format (with left-zero padding, as needed) when returned to the caller and automatically converted back to unsigned, zoned-decimal format in the VSAM record.  For example, this data type is useful for (among other things) zoned date and time fields. *But, packed date and time fields must use the* PACKED *data type.*  The COBOL equivalent is a PIC 9 field without a V position and USAGE IS DISPLAY. |

## **IMPLEMENTATION**

Create the five object decks, first, as standard 24-bit batch subroutines (actually, four subroutines and one main program—but as an object deck). Then, link this program's object deck as a standard, 24-bit batch main program (even though it switches to 31-bit addressing mode, internally).

### Table 2: Program Components

| **Source name** | **Description** |
| --- | --- |
| RXVSAMBK | RXVSAMIO — Build Key Buffer Subroutine |
| RXVSAMBR | RXVSAMIO — Build Record Buffer Subroutine |
| RXVSAMXA | RXVSAMIO — Extract Dataset Attrs Subroutine |
| RXVSAMXR | RXVSAMIO — Extract Record Fields Subroutine |
| RXVSAMIO | REXX Function To Access VSAM Files & Statistics |

This main program will not, however, then be available for just any program to call. Only a REXX procedure will be able to call it. Furthermore, do not distribute the linked main program because, in addition to its own subroutines, it also links in a couple of IBM subroutines that are not licensed for distribution.

## **INPUT**

On input, the following stem variable elements are used.

### Table 3: Input Stem Elements

| **Element name** | **Description** |
| --- | --- |
| ...\_rln | is the value for the record length to be used for variable-length record processing (\_rmx is the default). This would also be used for the logical record length when writing MSAM files. |
| ...\_rmx | is the value for the maximum record length (at OPEN time, this will be supplied automatically as a lookup from the associated catalog—it is highly inadvisable that this value should be changed). This would also be the maximum block size when processing MSAM files. |
| ...\_kln | is the value for the key length to be used for generic POINT or READ processing (\_kmx is the default). |
| ...\_kmx | is the value for the maximum key length (at OPEN time, this will be supplied automatically as a lookup from the associated catalog—it is highly inadvisable that this value should be changed). |
| ...\_key.0 | is the value specifying the total number of contiguous key fields that make up a full key for this file (for RBA or RRN access, specify a value of 1). |
| ...\_key.n | where n is from 1 to the value of \_key.0, in sequence and without gaps, this specifies one or more field definitions to make up a full key for this file; where its 5-word value is:  word1 is the actual name of this key field  (assembler conventions up to 32 characters),  word2 is data type of the field in the record,  word3 is the key zero-offset of this key field,  word4 is the byte length of this key field,  word5 is the number of decimal digits (up to 15)  for 'PACKED' or 'ZONED' key fields (but, must be  zero for 'IZONED', 'BINARY', and 'UBIN' key fields).  For RBA or RRN access, you must define a single 'UBIN'-type, 0-offset, 4‑byte, 0-decimal key field. |
| ...\_key.key\_name | contains the character or decimal value for the associated key field (where "key\_name" is one of those found in word1 of the \_key.n definitions given, undefined "key\_name" fields default to all binary zeroes, and supplied values are automatically converted by this function before storing in the file and before storing in the associated stem variables—such that the caller only has to deal with them in character and decimal formats, as appropriate). |
| ...\_fld.0 | is the value specifying the total number of contiguous fields that make up a full record for this file. |
| ...\_fld.n | where n is from 1 to the value of \_fld.0, in sequence and without gaps, this specifies one or more field definitions to make up a full record for this file; where its 5-word value is:  word1 is the actual name of this record field  (assembler conventions up to 32 characters),  word2 is data type of the field in the record,  word3 is the record zero-offset of this field,  word4 is the byte length of this record field,  word5 is the number of decimal digits (up to 15)  for 'PACKED' or 'ZONED' record fields (but, must be  zero for 'IZONED', 'BINARY', and 'UBIN' record fields). |
| ...\_fld.fld\_name | contains the character or decimal value for the associated record field (where "fld\_name" is one of those found in word1 of the \_fld.n definitions given and values are automatically converted by this function before storing in the file and before storing in the associated stem variables—such that the caller only has to deal with them in character and decimal formats, as appropriate). |

Note that if a non-binary field value on disk is low-values (all binary zeroes), then this function returns the \*NULL keyword as the associated variable value. Binary field values, of course, return a zero (0) as the associated variable value in this situation. Conversely, the caller may pass the \*NULL keyword, as a variable value, to set the associated field on disk to all binary zeroes regardless of the field's data type.

For complete functionality, then, if a non-binary field value on disk is high-values (all binary ones), then this function returns the \*HIGH keyword as the associated variable value. Signed binary field values, of course, return a minus one (-1) as the associated variable value in this situation. Conversely, the caller may pass the \*HIGH keyword, as a variable value, to set the associated field on disk to all binary ones regardless of the field's data type.

So, as an example, the following shows a sample COBOL record layout and, after that, what the associated RXVSAMIO data definitions for the key layout and record layout would look like.

### Example 1: Equivalent Data Definitions

|  |
| --- |
| ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--  01 WSINDX-RECORD.  03 WSIX-KEY.  05 WSIX-KEY-CORP PIC X(5).  05 WSIX-ID.  07 WSIX-ID-DATE PIC 9(8).  07 WSIX-ID-BATCH PIC 99.  03 WSIX-DATA.  05 WSIX-BATCH-TYPE PIC X.  05 WSIX-CREATED.  07 WSIX-CREATED-DATE PIC S9(9) PACKED-DECIMAL.  07 WSIX-CREATED-TIME PIC S9(7) PACKED-DECIMAL.  05 WSIX-CHANGED.  07 WSIX-CHANGED-DATE PIC S9(9) PACKED-DECIMAL.  07 WSIX-CHANGED-TIME PIC S9(7) PACKED-DECIMAL.  05 WSIX-PROCESSED.  07 WSIX-PROCESSED-DATE PIC S9(9) PACKED-DECIMAL.  07 WSIX-PROCESSED-TIME PIC S9(7) PACKED-DECIMAL.  05 WSIX-REC-COUNT PIC S9(7) PACKED-DECIMAL.  05 WSIX-SEG-COUNT PIC S9(7) PACKED-DECIMAL.  05 WSIX-USER-ID PIC X(8).  05 WSIX-COMPLETE PIC X(1).  05 WSIX-DELETE PIC X(1).  05 WSIX-FULL-AUDIT.  07 WSIX-AUDIT-IV PIC X.  07 WSIX-AUDIT-AR PIC X.  05 PIC X(27).  ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--  wsindx. = ''; /\* initialize file stem variable \*/  wsindx.\_key.0 = 3; /\* define key field layout \*/  /\* type offset leng decm \*/  wsindx.\_key.1 = 'WSIX\_KEY\_CORP' 'CHAR' 0 5;  wsindx.\_key.2 = 'WSIX\_ID\_DATE' 'IZONED' 5 8 0;  wsindx.\_key.3 = 'WSIX\_ID\_BATCH' 'IZONED' 13 2 0;  wsindx.\_fld.0 = 18; /\* define record field layout \*/  /\* type offset leng decm \*/  wsindx.\_fld.1 = 'WSIX\_KEY\_CORP' 'CHAR' 0 5;  wsindx.\_fld.2 = 'WSIX\_ID\_DATE' 'IZONED' 5 8 0;  wsindx.\_fld.3 = 'WSIX\_ID\_BATCH' 'IZONED' 13 2 0;  wsindx.\_fld.4 = 'WSIX\_BATCH\_TYPE' 'CHAR' 15 1;  wsindx.\_fld.5 = 'WSIX\_CREATED\_DATE' 'PACKED' 16 5 0;  wsindx.\_fld.6 = 'WSIX\_CREATED\_TIME' 'PACKED' 21 4 0;  wsindx.\_fld.7 = 'WSIX\_CHANGED\_DATE' 'PACKED' 25 5 0;  wsindx.\_fld.8 = 'WSIX\_CHANGED\_TIME' 'PACKED' 30 4 0;  wsindx.\_fld.9 = 'WSIX\_PROCESSED\_DATE' 'PACKED' 34 5 0;  wsindx.\_fld.10 = 'WSIX\_PROCESSED\_TIME' 'PACKED' 39 4 0;  wsindx.\_fld.11 = 'WSIX\_REC\_COUNT' 'PACKED' 43 4 0;  wsindx.\_fld.12 = 'WSIX\_SEG\_COUNT' 'PACKED' 47 4 0;  wsindx.\_fld.13 = 'WSIX\_USER\_ID' 'CHAR' 51 8;  wsindx.\_fld.14 = 'WSIX\_COMPLETE' 'CHAR' 59 1;  wsindx.\_fld.15 = 'WSIX\_DELETE' 'CHAR' 60 1;  wsindx.\_fld.16 = 'WSIX\_AUDIT\_IV' 'CHAR' 61 1;  wsindx.\_fld.17 = 'WSIX\_AUDIT\_AR' 'CHAR' 62 1;  wsindx.\_fld.18 = 'WSIX\_FILLER' 'CHAR' 63 27; |

Note that the last line of the data definition, above, is only required if you want to have the unused portion of your record initialized with something other than binary zeroes. The RXVSAMIO function implicitly takes care of that. But, by defining this area, you can explicitly initialize it yourself with spaces or, .e.g., with binary zeroes (using the \*NULL keyword, previously mentioned) or with binary ones (using the \*HIGH keyword, previously mentioned).

## **OUTPUT**

After every request, the following stem variable elements are returned—as available.

### Table 4: Output After Requests

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_ddn | is the data label name for the file (variable form). |
| ...\_lfn | is the name of the last access function requested. |
| ...\_lfm | is the name of the last failed macro or process, if any. |
| ...\_rc | is the VSAM Return Code (in decimal). |
| ...\_ec | is the VSAM Error Code (in decimal) or, if a catalog management error, then this is two words for the decimal Reason Code and the module id. |
| ...\_fc | is the VSAM Feedback Function Code (in decimal). |

After a successful OPEN, the following additional stem variable elements are returned.

### Table 5: Output After OPEN

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_dsn | is the data set name for the file. |
| ...\_typ | is the file type: KSDS, XLDS, ESDS, PATH, RRDS, VRDS, SAME, SAMI, or AIX. |
| ...\_rmx | is the maximum record length for the file (for MSAM files this is the maximum block size—see \_rln for the MSAM logical record length). |
| ...\_kmx | is the maximum key length for the file (for RBA and RRN files this is always four). |
| ...\_rkp | is the zero-offset relative key position in the record (for RBA and RRN files this is always zero). |

After a successful READ, the following additional stem variable elements are returned (the \_key.key\_name values are never changed by this function).

### Table 6: Output After READ

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_fld.fld\_name | these contain the character or decimal values for the associated record fields (each "fld\_name" is as found in word1 of the \_fld.n definitions given and values are automatically converted by this function as outlined, above). |
| ...\_rba | is the record's relative byte address (in decimal). |
| ...\_rrn | is the record's relative number (for RRDS, in decimal). |
| ...\_rln | is the length of the (possibly variable-length) record just retrieved. |

After the ATTRS request, the following six sets of stem variable elements are returned.

### Table 7: Output After ATTRS

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_ddn | is the data label name for the cluster, path, or AIX. |
| ...\_dsn | is the data set name for the cluster, path, or AIX. |
| ...\_catlbl | is the associated catalog data label name. |
| ...\_catdsn | is the associated catalog dataset name. |
| ...\_typ | is the file type: KSDS, XLDS, ESDS, PATH, RRDS, VRDS, SAME, SAMI, or AIX. |
| ...\_rmx | is the maximum record length for the file (or the maximum block size for MSAM files). |
| ...\_rav | is the average record length for the file (or the logical record length for MSAM files). |
| ...\_rfm | is the record format for the file (MSAM, only; F, FB, V, VB, or U). |
| ...\_kmx | is the maximum key length for the file (for RBA and RRN files this is always four). |
| ...\_rkp | is the zero-offset relative key position in the record (for RBA and RRN files this is always zero). |

### Table 8: Base Component after ATTRS

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_crea | is the creation date for the cluster, path, or AIX. |
| ...\_expd | is the expiration date for the cluster, path, or AIX. |
| ...\_tstp | is the last updated timestamp for cluster, path, or AIX (two words). |
| ...\_shro | is the share options for the cluster, path, or AIX (two words). |

### Table 9: Data Component after ATTRS

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_data | is the DATA component name. |
| ...\_dcsz | is the DATA control interval (CI) size. |
| ...\_dexc | is the number of DATA EXCPs. |
| ...\_dext | is the number of DATA extents. |
| ...\_dcis | is the number of DATA CI splits. |
| ...\_dcas | is the number of DATA CA splits. |
| ...\_dfsp | is the DATA CI/CA freespace requested (two words). |
| ...\_dhal | is the hi-allocated DATA RBA or XXL CI (in decimal). |
| ...\_dhus | is the hi-used DATA RBA or XXL CI (in decimal). |
| ...\_dtal | is the type of DATA allocation (CYL, TRK, or BLK). |
| ...\_dpal | is the primary DATA allocation. |
| ...\_dsal | is the secondary DATA allocation. |
| ...\_dvol | is the list of DATA volume(s). |

### Table 10: Index Component after ATTRS

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_indx | is the INDEX component name (if any). |
| ...\_icsz | is the INDEX control interval (CI) size. |
| ...\_iexc | is the number of INDEX EXCPs. |
| ...\_iext | is the number of INDEX extents. |
| ...\_ilvl | is the number of INDEX levels. |
| ...\_issr | is the INDEX sequence set RBA (in decimal). |
| ...\_ihlr | is the INDEX high-level RBA (in decimal). |
| ...\_ihal | is the hi-allocated INDEX RBA (in decimal). |
| ...\_ihus | is the hi-used INDEX RBA (in decimal). |
| ...\_ital | is the type of INDEX allocation (CYL, TRK, or BLK). |
| ...\_ipal | is the primary INDEX allocation. |
| ...\_isal | is the secondary INDEX allocation. |
| ...\_ivol | is the list of INDEX volume(s). |

### Table 11: Dataset Attributes after ATTRS

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_comp | is a Boolean value indicating a COMPRESSed dataset. |
| ...\_eras | is a Boolean value indicating the ERASE setting. |
| ...\_imbd | is a Boolean value indicating the IMBED setting. |
| ...\_ordr | is a Boolean value indicating the ORDERED setting. |
| ...\_rcvr | is a Boolean value indicating the RECOVERY setting. |
| ...\_repl | is a Boolean value indicating the REPLICATE setting. |
| ...\_reus | is a Boolean value indicating the REUSE setting. |
| ...\_span | is a Boolean value indicating the SPANNED setting. |
| ...\_ukey | is a Boolean value for the UNIQUEKEY setting (AIX, only). |
| ...\_upgr | is a Boolean value for the UPGRADE setting (AIX, only). |
| ...\_updt | is a Boolean value for the UPDATE setting (PATH, only). |
| ...\_wchk | is a Boolean value indicating the WRITECHECK setting. |

### Table 12: Dataset Statistics after ATTRS

|  |  |
| --- | --- |
| **Element name** | **Description** |
| ...\_recs | is the number of records in the file. |
| ...\_dels | is the number of deletes issued against the file. |
| ...\_adds | is the number of inserts issued against the file. |
| ...\_upds | is the number of updates issued against the file. |
| ...\_gets | is the number of retrievals issued against the file. |

In addition, the RXVSAMIO function also sets the REXX "RC" special variable with a value indicating the success or failure of the requested operation. You should analyze the "RC" variable's value for errors. Those values are as follows:

### Table 13: Return Codes

|  |  |
| --- | --- |
| **Return code** | **Description** |
| -9 | Record length error in stem (missing or out of range). |
| -8 | Key length error in stem (missing or out of range). |
| -7 | One or more of the \_fld elements are invalid (this could be either a \_fld.n element or the associated \_fld.field\_name element—see the note below). |
| -6 | One or more of the \_key elements are invalid (this could be either a \_key.n element or the associated \_key.field\_name element—see the note below). |
| -5 | Specified file label is for a NOCIFORMAT file. |
| -4 | Specified file label is missing a data component. |
| -3 | Specified file label is not a cluster, aix, or path entry. |
| -2 | A maximum of 9 files have already been used. |
| -1 | VSAM end-of-file or top-of-file occurred. |
| 0 | No errors occurred. |
| 1 | Request argument 1 is invalid. |
| 2 | Request argument 2 is invalid. |
| 3 | Request argument 3 is invalid. |
| 4 | Argument error—missing, too many, or too long. |
| 8 | Internal REXX function or system macro failed: also, the failing function's / macro's register 15 value is returned in the REXX "RESULT" special variable. |
| 12 | REXX environment not present. |
| 16 | VSAM error occurred (as noted under "OUTPUT" above). |

Note that if a subroutine of RXVSAMIO produces an error, then the variable "SRC" may be created to provide additional information for diagnostic purposes. For example, if "RC" is -6 or -7 then "SRC" is the \_key.n or \_fld.n element number, respectively, where the error was detected.

## **INVOCATION**

Invoking this external REXX function is as easy as you would invoke any other REXX function. Like other REXX functions, this can take the form of either an invoked function or a called subroutine. For example, the following are exactly equivalent.

|  |
| --- |
| result = SLEEP(1); /\* invoked as a function \*/  Call SLEEP 1; /\* called as a subroutine \*/ |

Meaning: When called as a subroutine, the returned result from a REXX function is not discarded but is automatically assigned to the REXX "RESULT" special variable.

That said, the following is an example of sequentially reading an entire file and displaying the fields (notice that, in this case, the key field names match the record field names but the offsets are different because they are relative to the beginning of the key and to the beginning of the record, respectively).

### Example 2: Sequential Access

|  |
| --- |
| myfile. = ''; /\* initialize stem variable \*/  myfile.\_key.0 = 2; /\* define key field layout \*/  myfile.\_key.1 = 'FIELD\_NAME\_2' 'CHAR' 0 10;  myfile.\_key.2 = 'FIELD\_NAME\_3' 'ZONED' 10 8 0;  myfile.\_fld.0 = 6; /\* define record field layout \*/  myfile.\_fld.1 = 'FIELD\_NAME\_1' 'BINARY' 0 8 0;  myfile.\_fld.2 = 'FIELD\_NAME\_2' 'CHAR' 8 10;  myfile.\_fld.3 = 'FIELD\_NAME\_3' 'ZONED' 18 8 0;  myfile.\_fld.4 = 'FIELD\_NAME\_4' 'CHAR' 26 30;  myfile.\_fld.5 = 'FIELD\_NAME\_5' 'PACKED' 56 7 2;  myfile.\_fld.6 = 'FIELD\_NAME\_6' 'BINARY' 63 4 0;  /\* open the dataset \*/  Call RXVSAMIO 'MYFILE', 'OPEN';  If rc = 0 Then Do /\* begin dataset retrieval \*/  Say "List of file fields:";  Say " ";  r = 0;  Do Until rc <> 0 /\* rc of -1 indicates end-of-file \*/  Call RXVSAMIO 'MYFILE', 'READ';  If rc = 0 Then Do  r = r + 1;  Do i = 1 To myfile.\_fld.0  pfx = 'Record' r', field' i',' Word(myfile.\_fld.i,1);  val = Value('myfile.\_fld. 'Word(myfile.\_fld.i,1));  typ = Word(myfile.\_fld.i,2);  Select  When val = '\*NULL',  | val = '\*HIGH' Then Say pfx "=" val;  When typ = 'HEX' Then Say pfx "= x'"val"'";  When typ = 'CHAR' Then Say pfx "= '"val"'";  Otherwise Say pfx "=" val;  End  End  Say " ";  End  End  End  /\* if no errors, close the dataset \*/  If rc = 0 | rc = -1 Then Do  Call RXVSAMIO 'MYFILE', 'CLOSE';  End  /\* report any errors \*/  If rc <> 0 Then Do  If rc <> 16 Then  Say "RXVSAMIO Call error: rc="rc", result="result",",  "ddname="myfile.\_ddn",",  "request="myfile.\_lfn",",  "macro="myfile.\_lfm;  Else Do  Say "RXVSAMIO" myfile.\_lfn "error --" myfile.\_ddn "--",  "rc="myfile.\_rc", ec="myfile.\_ec"," myfile.\_lfm;  End  Exit 4;  End |

...and, using the above key & field definitions and the error reporting, the following is an example of checking to see if a record exists before trying to delete it.

### Example 3: Random Access with Delete

|  |
| --- |
| /\* open the dataset for update \*/  CALL RXVSAMIO 'MYFILE', 'OPEN', 'update', 'random';  If rc = 0 Then Do /\* random retrieval of record \*/  myfile.\_key.FIELD\_NAME\_2 = "5712434-10"; /\* order number \*/  myfile.\_key.FIELD\_NAME\_3 = 12; /\* line number \*/  CALL RXVSAMIO 'MYFILE', 'READ', 'key';  End  If rc = 16,  & myfile.\_rc = 8, /\* not found? \*/  & myfile.\_ec = 16 Then  rc = 0; /\* allow "not found" \*/  Else Do  If rc = 0, /\* correct record key? \*/  & myfile.\_fld.FIELD\_NAME\_2 = myfile.\_key.FIELD\_NAME\_2,  & myfile.\_fld.FIELD\_NAME\_3 = myfile.\_key.FIELD\_NAME\_3,  Then Do /\* delete the record \*/  CALL RXVSAMIO 'MYFILE', 'DELETE';  End  End  /\* if no errors, close the dataset \*/  If rc = 0 | rc = -1 Then Do  Call RXVSAMIO 'MYFILE', 'CLOSE';  End |

...and, again (using the original field definitions and the error reporting, but with an RBA key definition), the following is an example of using an RBA to begin forward record processing. (Note that this same scenario would work for an RRDS dataset except that you would use an RRN key instead of an RBA address.)

### Example 4: Dynamic Access with RBA or RRN

|  |
| --- |
| myfile. = ''; /\* initialize stem variable \*/  myfile.\_key.0 = 1; /\* define the RBA address key \*/  myfile.\_key.1 = 'MYFILE\_RBA\_ADR' 'UBIN' 0 4 0;  /\* same record layout as before \*/  /\* open the dataset for input \*/  CALL RXVSAMIO 'MYFILE', 'OPEN', 'input', 'dynamic';  If rc = 0 Then Do /\* position for forward processing \*/  myfile.\_key.MYFILE\_RBA\_ADR = 7504; /\* RBA in decimal \*/  CALL RXVSAMIO 'MYFILE', 'FWDPOS', 'rba';  End  If rc = 0 Then Do /\* begin dataset retrieval \*/  Say "List of file keys:";  Say " ";  r = 0;  Do Until rc <> 0 /\* rc of -1 indicates end-of-file \*/  Call RXVSAMIO 'MYFILE', 'READ';  If rc = 0 Then Do  r = r + 1;  Say "Record" r", rba =" myfile.\_rba",",  "key =" myfile.\_fld.FIELD\_NAME\_2,  myfile.\_fld.FIELD\_NAME\_3;  End  End  End  /\* if no errors, close the dataset \*/  If rc = 0 | rc = -1 Then Do  Call RXVSAMIO 'MYFILE', 'CLOSE';  End |

...and, finally (using the original key & field definitions and the error reporting, above), the following is an example of starting at the end of a key range and reading backwards to process all records in the key range in a generic manner. If you have an exact (full) key (or if the entire key is high-values—for retrieving the last record in the file), then you can just start there with BWDPOS. Otherwise, you need to use FWDPOS to set up to get the next full key and then use BWDPOS to proceed backwards from there—as demonstrated below.

### Example 5: Backward Processing of a Key Range

|  |
| --- |
| desired\_order\_no = "5712434-10"; /\* process this order \*/  /\* open the dataset for input \*/  CALL RXVSAMIO 'MYFILE', 'OPEN', 'input', 'dynamic';  If rc = 0 Then Do /\* point to next key range \*/  myfile.\_key.FIELD\_NAME\_2 = desired\_order\_no;  myfile.\_key.FIELD\_NAME\_3 = '\*HIGH'; /\* end of range \*/  CALL RXVSAMIO 'MYFILE', 'FWDPOS', 'key', 'gteq';  End  If rc = 0 Then Do /\* get record from next key range \*/  Call RXVSAMIO 'MYFILE', 'READ', 'next';  If rc = 0 Then Do /\* reverse file direction \*/  myfile.\_key.FIELD\_NAME\_2 = myfile.\_fld.FIELD\_NAME\_2  myfile.\_key.FIELD\_NAME\_3 = myfile.\_fld.FIELD\_NAME\_3  CALL RXVSAMIO 'MYFILE', 'BWDPOS', 'key';  End  If rc = 0 Then Do /\* get same record but in reverse \*/  CALL RXVSAMIO 'MYFILE', 'READ', 'prev';  End  End  If rc = 0 Then Do /\* get the desired key range \*/  Say "List of file keys:";  Say " ";  r = 0;  Do Until rc <> 0 /\* rc of -1 indicates end-of-file \*/  Call RXVSAMIO 'MYFILE', 'READ', 'prev'  If rc = 0,  & myfile.\_fld.FIELD\_NAME\_2 <> desired\_order\_no,  Then Do  rc = -1; /\* end of desired key range \*/  End  If rc = 0 Then Do  r = r + 1;  Say "Record" r", key =" myfile.\_fld.FIELD\_NAME\_2,  myfile.\_fld.FIELD\_NAME\_3;  End  End  End  /\* if no errors, close the dataset \*/  If rc = 0 | rc = -1 Then Do  Call RXVSAMIO 'MYFILE', 'CLOSE';  End |

Lastly, note that the full key fields (layout) definition is always required. But, the full record fields (layout) definition is only required when writing or updating a file record. If just reading a file, then only the record fields that will be used need to be defined. However, even for a partial definition, the \_fld.n stem element numbers \*must\* be in sequence with \*no\* gaps.

— Dave Clark