VsamEx[treme]™

The Original "un-database"

(With SHA+ Encryption)

User Guide

February 2011

Windows & Linux Edition's

Software Source

PO Box 23306
San Jose, CA 95153
United States of America

Email
software_src@earthlink.net

Web
www.1-software-source.com

Limited Warranty

Software Source provides the **VsamEx[treme]™** software and accompanying materials with the following limited warranty:

When used in accordance with instructions, Software Source warrants this product against any defects due to faulty materials or workmanship, for a period of sixty days from the purchase date. If Software Source receives notification within this warranty period of defective materials or workmanship, and determines that such notification is correct, then Software Source will replace the defective product distribution and/or documentation at no charge. This warranty does not cover damage due to accident, abuse, misuse, or improper installation of the product. Software Source authorizes no other warranty, written or oral, and there are no implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. In no event will Software Source be liable for any damages, including any general, special, incidental, indirect, exemplary, or consequential damages arising out of the use, misuse, or inability to use the **VsamEx[treme]™** product. The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective product distribution and/or documentation and shall not include or extend to any claim for or right to recover any other damages, including, but not limited to, loss of profit, data, or use of the software, or special, incidental, or consequential damages or other similar claims, even if Software Source has been specifically advised of the possibility of such damages; in no event, however, will Software Source's liability exceed the actual amount paid for the product.

NOTICE

Any use of VsamEx[treme]™ or any of its components constitutes acceptance of the terms and conditions included in the VsamEx[treme] License Agreement and its Limited Warranty.

Our Special Thanks for contributions to this project go to:

❖ ***Jay J. Falconer*** *at Bitwise Software International, Inc., http://www.shoppingQ.com, for his help in testing VsamEx for Linux, his many suggestions and his creation of a VsamEx C++ class wrapper that ships with the product.*

❖ ***Mike Whittingham*** *at Chaos Software, Inc., http://www.chaossoftware.com, for his long time support, kind critical comments and meaningful suggestions.*

Visit their websites to see real world applications of VsamEx[treme] and its predecessor VB/ISAM!

Tony Altwies
Software Source

CONTENTS

Copyright Notice and Trademarks

Software License Keys

VsamEx License Keys, software, and documentation are the sole property of **Software Source**. Said property is licensed not sold. **Software Source** retains sole title and rights to all said property except rights specifically granted to others by **Software Source**. You may purchase and register rights to use a **License Key** directly from **Software Source**. **Software Source** hereby licenses the use of **VsamEx** for the purpose of software development, provided all of the following conditions are met:

1. Only one person at a time on one machine may use a particular **License Key** for the purpose of developing applications linked to **VsamEx**. 2. You must use the **License Key** provided to you by **Software Source**. 3. You must register the transfer, if any, of the **License Key** with **Software Source**. 4. You may not enable unregistered use of **VsamEx** in a development environment. 5. You may not use a **License Key** already registered to someone else. 6. **Software Source** must have a record of originally issuing the **License Key**.

As long as all of the above conditions are met, you are expressly granted the rights to copy, transfer, and distribute your applications linked with the **VsamEx Library** as part of the software you develop using **VsamEx** without Royalty**.**

Scope

This manual describes the capabilities and mechanisms of **VsamEx** versions for Linux and Windows operating systems. The manual includes the latest features and hints for optimizing your use of **VsamEx** in the design of your multi-user network software products. A complete description of all function can be found in the ***Programmers Reference manual***.

Overview

VsamEx may be used in a standalone mode or in a multi-user, shared networked mode with full record locking functions, **VsamLock** and **VsamUnlock.** This extends the **VsamOpen** function with two additional shared file access modes.

The remainder of the **Introduction** section discusses **MultiUser** and **SingleUser** file and program compatibility, issues for network environments, and file access modes. *File Access Modes*: In addition to the READ_ONLY and READ_WRITE access modes provided by the **VsamOpen** function, READ_ONLY_SHARED and READ_WRITE_SHARED modes are supported for Multiple simultaneous user access. This section describes the use of all four **VsamOpen** modes.

The **Data Access Coordination** section describes two **MultiUser** forms of data access coordination; automatic locking mechanisms to preserve file integrity, and two functions -- **VsamLock** and **VsamUnlock** -- to preserve application integrity. This section describes these mechanisms, and discusses index pointer positioning.

VsamEx Compatibility

All datasets created by VB/ISAM may be accessed by VsamEx. Once a Dataset is opened in a READ_WRITE non-shared mode (mode 1) by VsamEx, it is no longer compatible with VB/ISAM access. While the API's are similar, the extended features of VsamEx preclude 100% backward compatibility. The conversion process is automatic only in this mode. VsamEx will create a set of VsamEx Field Definitions corresponding to the format string in the VB/ISAM dataset. The Format String will be removed from the Notes section of the dataset. Notes will also be converted to Dictionary records with a prefix of "-Notes-" (see Dictionary below).

The default name for Fields will be F[n][ffd] where n = the field number and ffd = the original field format definition as found in the old format string. So, a format string of "$,a3%,#" would generate "F1$", "F2a3%", "F3a3%", "F4a3%", "F5#" for default field names. As an alternative, an .isc file may be created with each line representing a field. Remember the record Key is first and is field 0. Type will correspond to the format string element: "$,$,#"

NewID,31,L	**Field name, width, Justification**
Name,20,L	**Field name, width, Justification**
Salary,15,R	**Field name, width, Justification**

Field width and justification are only for reference and do not restrict the length of data in string fields. For example, these attributes may be referenced by a report module.

VsamEx is supported as a DLL, OCX and library (both Link .lib & .dll). A License Key, provided by Software Source, will enable you to open any given dataset. VsamEx will function for about 14 days without a valid license. All current versions of **VsamEx** are ***file compatible*** with all previous version of VsamEx. Once an older VB/ISAM dataset is converted or a new one is created by VsamEx, it is no longer accessible by VB/ISAM DLL's. The only version of VsamEx is the **Enterprise Edition** of **VsamEx -** allowing datasets to grow to over 2.1 GB.

The application programmer's interface (API) is similar to previous versions of VB/ISAM; all function definitions reflect similar parameters except that they are geared toward a newer more efficient interface allowing more features and easier integration into Linux "C/C++" as well as Windows .NET, "C/C++", Visual Basic, Pascal. basically any language capable of calling a DLL or link with a standard library.

Network Considerations

VsamEx shared file access uses the file locking features supported in all Linux as well as Windows operating systems, and is not designed around the specific workings of any given type of networking software. This means that any network software that properly supports **SHARED** access or provides equivalent functionality, will be compatible with **VsamEx shared file** access.

File Access Modes

» *The* **VsamOpen** *function, in access modes 2 and 3* (READ_ONLY_SHARED *and* READ_WRITE_SHARED), *will return* VIS_ACCESS_DENIED *if you try to open a dataset located on a disk that is not share enabled. Modes 0 and 1 continue to operate as in* previous versions of **VsamEx**. *Your application must also have network "privileges" to create, read, and write files on server or remote peer systems to use access modes 2 and 3.*

The **VsamOpen** function in **VsamEx** provides four file access modes:

mode 0	=	READ_ONLY
mode 1	=	READ_WRITE
mode 2	=	READ_ONLY_SHARED
mode 3	=	READ_WRITE_SHARED

Mode 0: A program can successfully open a file in mode 0 only if ***no*** other programs have the file open in mode 1 or mode 3; otherwise **VsamOpen** will return VIS_ACCESS_DENIED. In mode 0 access, the updating functions (**VsamPut**, **VsamDelete**, **VsamWriteNote, VsamWriteDict**, and **VsamFlush**) are disabled; they will all return VIS_ACCESS_DENIED for a dataset opened in mode 0. Note that a file can have multiple concurrent mode 0 and mode 2 accesses.

» Use mode 0 in your application when you want read only access with no chance of other processes acquiring update access.

Mode 1: A program can successfully open a file in READ_WRITE mode only if no other accesses to this file exist; otherwise, **VsamOpen** will return VIS_ACCESS_DENIED.

» Use mode 1 in your application to exclude all other processes from all access.
<u>» This mode is the only mode that will allow conversion of older VB/ISAM datasets.</u>

Mode 2: A program can successfully open a file in READ_ONLY_SHARED mode only if no other program has the file open in mode 1. In mode 2 access, the updating functions (**VsamPut**, **VsamDelete**, **VsamWriteNote**, **VsamWriteDict** and **VsamFlush etc.**) are disabled; they will all return VIS_ACCESS_DENIED. A file can have multiple concurrent mode 2 accesses, mode 0 and mode 2 accesses, or mode 2 and mode 3 accesses. Mode 0 and mode 3 are mutually exclusive.

» *Use mode 2 in your application when you want read-only access for a process when other processes may be updating the file (using mode 3); the locking mechanisms described in the next section are provided for coordination under these circumstances.*

Mode 3: a program can successfully open a file in READ_WRITE_SHARED mode only if no other program has the file open in mode 0 or mode 1. Note that a file can have multiple concurrent mode 2 and mode 3 accesses.

» *Use mode 3 in your application when you want read/write access to a file in coordination with other processes. The locking mechanisms described in the next section provide that coordination.*

Note that the only purpose of mode 2 is to provide convenient way to preclude "accidental" use of **VsamPut**, **VsamDelete**, **VsamWriteNote**, **VsamWriteDict**, and **VsamFlush** in complex coding situations. All access combinations may be constructed from modes 0, 1, and 3.

In summary, the following access mode combinations are compatible for concurrent access to the same **VsamEx** file: (The second and third combinations are functionally equivalent to the first, but are slightly slower because all mode 2 reads must check for potential mode 3 updates.)

mode 0 (READ_ONLY)	and mode 0 (READ_ONLY);
mode 0 (READ_ONLY)	and mode 2 (READ_ONLY_SHARED);
mode 2 (READ_ONLY_SHARED)	and mode 2 (READ_ONLY_SHARED);
mode 2 (READ_ONLY_SHARED)	and mode 3 (READ_WRITE_SHARED);
mode 3 (READ_WRITE_SHARED)	and mode 3 (READ_WRITE_SHARED);

... and exclusive access by a single process in mode 1 (READ_WRITE).

In the SHARED file access modes - mode 2 (READ_ONLY_SHARED), and mode 3 (READ_WRITE_SHARED) - **VsamEx** provides two forms of locking to coordinate multiuser access: Automatic locking and **VIS_BUSY**, and **VsamLock** and **VsamUnlock** semaphore functions.

DATA ACCESS COORDINATION

Automatic Locking and VIS_BUSY

First, to preserve dataset integrity, **VsamEx** automatically coordinates all functions that either change the dataset (**VsamPut**, **VsamDelete**, and **VsamWriteDict**) or read the dataset (**VsamGet**, **VsamReadDict**, and **VsamInfo**). All these functions are designed to interact without conflict in any combination, and to read or write consistent data.

Note that in mode 3, the **VsamFlush** *function becomes a "no-op", since the updating functions (***VsamPut***,* **VsamDelete***, and* **VsamWriteDict***) all perform an automatic internal flush-to-disk.*

VsamEx uses an intelligent step-wise locking scheme in these functions to deny access to other processes for as short a time as possible. Nevertheless, pathological situations are theoretically possible, in which a large number of network users simultaneously attempt access to the same part of a file. Since it's impossible to predict how long a delay may be required to gain access. All of these functions are designed to return **VIS_BUSY,** after having passed through the step-wise locking process, if they continue to be blocked from access.

» *In the* SHARED *file access modes, you* MUST *write your code to accommodate the possibility of a VIS_BUSY return from* VsamPut, VsamGet, VsamDelete, VsamWriteDict, VsamReadDict, *and* VsamInfo. *The recommended technique is to Sleep briefly and then try again. It is still possible get a VIS_BUSY return even if you've bracketed your access with explicit* VsamLock *and* VsamUnlock. *The main reason for this is that different records with different primary keys may exist in the same Group. If they are each accessed and then both written by different applications (or threads) at the same time, only one of the apps will momentarily receive permission to update the group. While VsamEx minimizes this wait time, a large number of simultaneous writes to the same Group may encounter situations where the wait time for permission queue insertion exceeds the time-out values.*

In some circumstances, these automatic locking mechanisms may be all you need. For example, if a number of concurrent users simply add new records to the same dataset, your application can just call **VsamPut** with the ADD_ONLY update mode, and then check for a VIS_UPDATE_VIOLATION return code to see if another user had already added the same record. Of course, you must still check for VIS_BUSY and retry if necessary.

Semaphores: VsamLock and VsamUnlock

Second, for explicit program control over multiuser access, **VsamEx** provides two functions: **VsamLock** and **VsamUnlock**. These functions allow a process to post and remove public notices (called semaphores) associated with a given dataset, that are visible to other processes that have that dataset open.

» Caution: Despite its name, **VsamLock** does not directly prevent record access. *Please read the rest of this section for a complete description of the process.*

You call **VsamLock** with only two arguments: the DatasetNumber& of an open dataset, and a "lock string". The call is successful if and only if no other process currently holds a lock on this particular string for this dataset.

Although most programmers will use primary index keys (i.e., record identifiers) as lock strings, this isn't required; *the lock string argument can be any text string, including the null string*. While most programmers use **VsamLock** and **VsamUnlock** to accomplish record locking, these functions really provide a broader, more flexible type of inter-process coordination. If you want to use **VsamLock** for record locking - to explicitly protect a given record from change - then you should call it with that record's primary index key as the lock string, and of course consistently follow lock/unlock conventions in your program logic, as outlined in the example.

» Note: Since programmers commonly use primary index keys as lock strings for record locking, **VsamEx** treats all lock strings case-insensitively, just as it treats index keys. For example, "JonEs" and "jones" are considered identical. Similarly, **VsamEx** normalizes lock strings in the same way it does index keys, trims leading and trailing spaces and compacts multiple adjacent spaces to a single space. Lock strings must not exceed 255 bytes.

By itself, use of **VsamLock** with a primary index key does **NOT PREVENT** another process from doing whatever it wants with that record. Instead, the intent is that you'll use these semaphores as coordinated "program logic gateways" to **COOPERATIVELY SYNCHRONIZE** access to sensitive operations.

Suppose you want to examine an existing record and perhaps update it, depending on its contents. Your coding outline might look like this classic example of record locking:

Step 1: Determine the primary key of the record and call **VsamLock** for this dataset with the lock string set to that primary key. Examine the return code: If the function returned VIS_ACCESS_DENIED, some other process interested in this record got to this step first; wait . If the function returned VIS_OK, then we got here first for this record, and we can proceed. Note that other processes might now be processing other records, but we don't care; our record is "*protected*".)

Step 2: Call the **VsamGet** function to retrieve the record, and make your decisions about if and how to update it. If you decide not to do any updating, go to step 4.

Step 3: Make your changes to the data and call **VsamPut** save the changes.

Step 4: Call **VsamUnlock** for this dataset on the lock-string you locked in step 1.

» Note: Standard technique in any multiuser locking design is to ***minimize*** the time that any lock is held. If the decision-making stage of the above logic sequence involves user interaction, then the step 1 logic should be rewritten to abandon this access with a user message, rather than just wait; you don't want to keep someone hanging because someone else went on a coffee break in the middle of a lock sequence. Likewise, applications should not be coded such that locks are set and held while user input is required without timeouts.

The point of this example was that while your program was examining data in step 2, *no other process was changing that record* (since, given that you got to it before they did, they waited in their step one operation until you finished your step 4); when you did your step 3 updating, you could be confident that you were dealing with consistent data. *Again, since semaphore locks depend on consistent use, the above statement is true* ***only if*** *all programs accessing the file use this same "gateway" logic*. Because of the widely different criteria used in applications, we leave this choice and responsibility up to the application programmers.

To consider a more interesting example, suppose your application always does record access through lookups in a secondary index -- for example, a LastName index of an employee file. Now, by locking the string "Smith", you could protect access to *all* the Smiths (again assuming consistency of design in your application). This use of semaphores is an example of "class locking", allowing you to protect access over a range of data in one operation. More creative uses are certainly possible, including combinations of class locks and record locks, and any other kind of processes that require brief rapid communication, since these semaphores are ***generic network global signals***.

VsamLock allows a given process to concurrently hold up to 32 different semaphore locks on the same dataset. Most applications shouldn't require more than one or two.

If you want to implement a "file lock", establish the *convention* that some unique lock string represents that meaning. The null string is a good choice, since index keys cannot be null.

When a process closes a dataset with **VsamClose**, or ends (for any reason, including a crash), the network operating system will automatically release all semaphore locks that processes may not have unlocked.

VsamLock and **VsamUnlock** are designed for *coordinating* shared access updates, in which several processes have the same dataset open in access mode 3, READ_WRITE_SHARED (or perhaps a combination of modes 2 and 3). They will also work, however, in modes 0 and 1. Syntax and usage follow.

DATA ACCESS COORDINATION

General Notes on Locking

If process **A** wants to lock Smith and Jones (in that order) before proceeding, and process **B** wants to lock Jones and Smith (in that order) before proceeding, there is a potential for a logical problem commonly called a "deadly embrace", or "deadlock", if the timing sequence is just right. In this example, if process **A** locks Smith, but then process **B** locks Jones, neither process can proceed.

This is a universal problem not confined to **VsamEx**; the best way to avoid it is to unlock all previous locks in reverse sequence and start over if you can't get a lock. Alternatively, you may be able to design your logic to not require more than one concurrent lock within the same category of lockable objects.

DATA ACCESS COORDINATION

Index Pointer Positioning

Index pointers are *process-local* -- that is, each process with concurrent access to the same dataset maintains its own set of **VsamEx** index pointers. Index pointer positioning is based on **key values**, **not record position**. If, in a shared environment, you set an index pointer to a certain point -- for example, in front of (not on) the first Smith -- and another process adds more Smiths, you're still in front of the first Smith, even though that may be a different Smith than it used to be. Similarly, if you call **VsamEOF** in an index and another process adds a new last entry, you'll get that new last entry when you later call **VsamGet**/XPREVIOUS. Each index pointer is separately maintained and independent of other index pointers. Changing the index pointer position for one index does not change any other index pointer positions, including the primary index pointer.

Encryption

VsamEx supports full data encryption through the use of extended create and open functions (**VsamCreate** & **VsamOpen**). An encryption key may be any arbitrary, variable length string of characters, excluding character 0. Once a dataset is created using an encryption key, it can only be opened, read and written using the same exact key that it was created with. The encryption is done using a modified ultra high speed Secure Hashing Algorithm (SHA). The cost in performance can vary depending on the record size and group size. We have estimated it to be approximately 4% loss in performance when using encryption.

The encryption/decryption is performed in such a way that VsamEx data moves from host to client fully encrypted. This makes it very useful for moving data around any unsecured network.

If you are unfortunate enough to loose track of any of your encrypting keys, the data contained within an encrypted VsamEx dataset may not be retrievable*!!

YOU HAVE JUST COMMITTED "DATASIDE"!!

YOUR DATA JUST BIT[sic] THE DUST!!

YOU HAVE OFFICIALY CONTRIBUTED TO THE GREAT ENTROPIC HEAT DEATH OF THE UNIVERSE!!

** Even we at Software Source, with our ultra fast quark computers and special gluon grease, have found it necessary to travel back in time in order to recover lost encryption data! So far, the only individual we know of that can afford these services works at Microsoft!*

Sparse Record Fields

The new low level VsamEx record format allows for “sparse record fields”. By this we mean that not all records need have any or all of the fields defined in the dataset, e.g. Record # 1 only contains fields 3 and 4, while #2 only fields 1 and 9. Records that do not contain a field reserve zero (0) space for that field. When a record is read and a field is fetched that does not contain the field requested, the function will return a VIS_NOT_FOUND for that field.

When a dataset is created using **VsamCreate**, the old method of passing a record format string that defines a homogeneous record set is no longer part of that process. Instead, there is now a set of separate functions that are used to manage fields:

VsamAddField
VsamEnumAttribValues
VsamEnumFieldAttrib
VsamGetFieldAttribute
VsamSetFieldAttribute
VsamDeleteField
VsamPurgeField
(see API summary below or the VsamEx[treme] Programmers Reference)

There are certain default field attributes like Fnam, Fnum, Ftyp, Fwid, Fjst, Find, etc. that are always available. The applications programmer may now add/delete/modify new fields and attributes on the fly. Certain field attributes may not be modified such as “Fnum” and “Ftyp” and there must always be a “Primary” field defined (at Create time) for the dataset. While the default name for the primary is “Primary”, it may be changed by the application software.

Once a field Type is set, it must remain consistent over all records. The only way to change it is to delete the field and create a new one. Field Numbers are not intended to represent relative field position in the record. They are only used internally to permanently identify a field of a specific type. Its really the unique field ID. The way the new low level records are constructed, fields may be attached to a record in any order.

FEATURES & DESIGN

DATASET EXTENSIONS

The VsamCreate, VsamOpen & VsamKill functions accept Dataset names with extensions. The first two characters of the extension passed will be used to construct the extensions of related VsamEx files.

Example: "MyData.NEW" Would correspond to the VsamEx files ending with ".NE**D**", ".NE**M**", and ".NE**L**". The last character identifies the specific part, i.e. "**D**" is the Data, "**M**" is the Map, and "**L**" is the lock file. If no extension is specified the defaults will be ".VOD", ".VOM" & ".VOL".

NETWORK TUNING

Every computer running applications using **VsamEx**, may optionally have a parameter file named **VSAM.INI**. The file contains adjustable parameters under the "*[Options]*" heading. If VSAM.INI does not exist, *LockRetries* defaults to 255, *TimeDelay* defaults to 5, and *MinDelay* defaults to 5. Since some values in the **VSAM.INI** file can cause **VsamEx** to run slower, we recommend you do not create the file unless a dataset has many simultaneous users and is experiencing major lock contention problems. A very small number of installations need this file.

LockRetries is the number of times **VsamEx** will try to lock the map before returning VIS_BUSY. Map locking is required in some circumstances to ensure data integrity and consistency. It is then the responsibility of the application programmer to either try again or take some other action.

There is no wait if the lock is not busy. *TimeDelay* is the number of milliseconds to wait to retry if the lock is busy. If the lock is still busy, the wait time of the next retry is cut in half. This continues until the wait time reaches *MinDelay* milliseconds. All subsequent waits are at least *MinDelay* milliseconds each, until the retry count is exhausted. Once a lock is achieved, the retry time returns to the *TimeDelay* value. Larger values for *TimeDelay* tend to reduce lock contention for some large networks (25 workstations +). For very high speed (100>mhz Pentiums with PCI network cards) systems, the optimum value for *TimeDelay* may be above 256 with *MinDelay* at 1. If you experience significant lock contention, as evidenced by some systems never getting map lock access, experiment with *larger* values for *TimeDelay*. The general rule to follow is:

Use the smallest values that work appropriately for your system! For applications with more than 25 simultaneous users, we found the following values to work well:

```
[Options]
LockRetries=255
TimeDelay=256
MinDelay=1
```

DICTIONARY

VsamEx supports the concept of a dictionary. While VB/ISAM supported the Notes feature with the VsamReadNote and VsamWriteNote functions, VsamEx only supports the dictionary concept and will convert existing Notes to Dictionary records with a key prefix of “-Notes-“. So, after a conversion, a note whose key was “VersionNumber” in VB/ISAM will become “-Notes-VersionNumber” in VsamEx.

FEATURES & DESIGN

The Dictionary section may contain any number of records and each record may be up to 64k bytes in length. The records are unstructured strings and may contain arbitrary binary data. Each Dictionary record is identified with a Key string of up to 252 characters in length. All dictionary entries are maintained in sorted sequence and the only valid ways to read in the dictionary are XLOOKUP, XPREVIOUS and XNEXT. Data is always returned with Dictionary reads. Dictionary records, unlike Notes, are stored in the main data file (.isd). The only limit to the number of records is the maximum size of your data set, i.e. you may have a dataset consisting entirely of Dict type elements (records).

A set of example code is provided at the end of the VsamWriteDict API definition.

NOTES

1. The underscore character "_" must be used with care if used in an index or Primary. Odd behavior has been reported due to the following circumstance:

the letter "A" is hex 0x41 (0100 0001)
the letter "_" is hex 0x51 (0101 0001)
the letter "a" is hex 0x61 (0110 0001)

When testing keys for sequence, VsamEx uses functions that perform case insensitive string compares. They do this by or-ing 0x20 with all "alpha" characters. This will convert "A" to "a" but leave "_" unchanged. This means that even though the letter "A" is less than "_", which is less than "a" in true sort order, VsamEx will treat "A" and "a" as greater than "_".

Straightforward, simple API:

VsamCreate	Creates a new, and empty, dataset.
VsamKill	Deletes any dataset.
VsamFlush	Called in <u>non</u> shared modes to flush the last update to the disk.
VsamOpen	Opens a dataset for access.
VsamClose	Closes a dataset.
VsamPut	Adds or rewrites a record.
VsamDelete	Removes a record.
VsamGet	Both maneuvers through an index -- using Lookup, Next, Previous, and Current modifiers -- and optionally retrieves the data record.
VsamFreeRec	Free the Memory buffer allocated by a ***VsamGet.***
VsamBOF	Positions to the beginning of an *index.*
VsamEOF	Positions to the end of an *index.*
VsamMovePtr	Re-position a virtual index pointer.
VsamCancel	Used to cancel any secondary operation such as ***VsamMovePtr.***

VsamWriteDict	Write a record to dictionary portion of the dataset.
VsamReadDict	Read a record from dictionary portion of database.
VsamDeleteDict	delete a record from dictionary portion of dataset.
VsamSetDictBof	Move dictionary record pointer to BOF.
VsamSetDictEof	Move dictionary record pointer to EOF.
VsamSearch	Create a flat file or DynaSet of records.
VsamInfo	Retrieve database status information.
VsamLock	Lock a text Semaphore in the database.
VsamUnlock	Unlock a text Semaphore in the database.
VsamAddField	Dynamically adds a new record field to an existing dataset.
VsamEnumAttribValues	Returns a comma delimited string containing a list of all the values for the specified field attribute.

VsamEnumFieldAttrib	Returns a comma delimited string containing a list of all field attributes currently defined for this field.
VsamGetFieldAttribute	Read the value of a named field attribute.
VsamSetFieldAttribute	Set the value of a Field attribute or add a new attribute.
VsamDeleteField	Flags a field as having been removed from the dataset. As records are added/updated in the dataset, this field is no longer added. Calls to store field will return an error for this field definition.
VsamStoreField	Store data into a record field.
VsamFetchField	Fetch data from a record field.
VsamVal	Rapidly validate a dataset
VsamRebuild	Construct a new Dataset, salvages all primary records, re-indexes them, and compacts the dataset to its Minimum size.

Please See the ***VsamEx[treme]*** *Reference manual for detailed function descriptions.*

RECORDS, KEYS, & INDEXES

Records and fields: The Fundamental unit of a **VsamEx** file access is a "record." The **VsamPut** function writes a file record from the contents of a record buffer (GSTR) in your program, and the **VsamGet** function reads a record from the file into a similar (or the same) record buffer. Each "field" in a record may be one of the basic numerical data types that are 2 bytes, 4 bytes and 8 bytes in length. Additionally, string fields may be defined that are variable length and may contain any kind of binary data, not just text – all within the record-size maximum of **64KB.**)

Primary Keys: You must supply a "primary key" with every record you write. Primary keys are strings (variable-length - max 252 bytes) that *uniquely* identify the associated record in the file; if you try to write a new record with the same primary key as a record already in the file, **VsamEx** will replace the old record with the new one - although you can protect yourself from doing this inadvertently by using the **XADD_ONLY** modifier. Likewise the application may limit the update with **XREPLACE_ONLY.** *The default is both add and replace.* All of the Primary keys together are called the primary index. ***The Primary Key is considered to be record field number 0 and the Primary index is index 0.***

The **VsamCreate** function will automatically create the “Primary” field definition to initialize the Data Definitions. ***It is always Indexed. Its name may be changed but its field number (0) and Indexed status (“T”) can not be changed.***

All subsequent Field definitions are created by calling **VsamAddField**. Each field defined has several sets of attributes that are Predefined (see the Programmers Reference) and others that the user may add using the **VsamSetFieldAttribute** function. All attribute values for all fields are stored in a special dictionary record in the database and therefore its size is limited to 64k bytes. ***So if you create fields that have attributes totaling 1000 bytes each, you may only have room in this record for a maximum of 64 fields.***

Secondary Keys and Indexes: After you create a **VsamEx** file, you define additional fields by executing the **VsamAddField** function. Secondary keys, sometimes referred to as cross-reference keys, are the contents of string fields that have the “Index” attribute in its field definition set to “Y”. The indexed attribute may only be set to “Y” within the first 150 fields defined. Each “Indexed” field will affect overall performance when writing records. The more Indexes, the more time is required to update since each will cause a disk write.

Duplicate Keys: In contrast to primary keys, the keys in a secondary index need not be unique. For example, if you have a LastName Index for an employee file, you may have several "Smiths". We call these "duplicate keys". Duplicate-key entries in a secondary index are ordered in ascending primary-key order. For example, if the primary key is SocialSecurityNumber, the first "Smith" in the LastName index will be the one with the lowest-sorting Social Security Number.

Sparse indexes: Every record needn't be represented in every secondary index; in other words, not all fields need to be present in all records. This gives you a "sparse" (or "occasional") index capability, so you can keep quickly traversed, small lists of various kinds of "special" records. (This is also the only practical way to approach lots of indexes in a given record; even for **VsamEx**, updating so many indexes when you write or delete such a record would take a lot of time.)

You may not, by the way, write the null string for a primary key.

More About Keys

Maximum Key Lengths: *Although all secondary index keys can be variable length strings, the maximum key length of the lowest level binary record is 253 bytes. The length of the primary key plus the length of the secondary field used to form an index key must be less than or equal to 252 characters. For instance, if you had a 50-byte primary key, the largest maximum secondary you could store would be 202 bytes. For the fastest access with large files, keep your keys short, especially the primary.*

Case-insensitive Keys: **VsamEx** protects your dataset from keyboard-error sequencing and duplication problems by treating upper- and lower-case letters as identical when comparing keys; thus "Jones" and "jones" are considered identical. If they're primary keys, only one of them can therefore exist in the dataset. Specifically, **VsamEx** doesn't convert characters, but sorts and compares upper-case letters as if they were lower-case: ASCII 65 is treated the same as ASCII 97.

Key Normalization: Another form of error protection is the normalization of keys: **VsamEx** trims leading and trailing spaces from the internal representation of keys. Furthermore, it collapses any multiple embedded spaces into a single space (**VsamEx** leaves the actual data fields alone.) **VsamEx** also normalizes key-search function parameters, such as the *Selector* parameter in VsamGet/Lookup. In this way, a lookup on A-B-space-C will successfully find a key that was originality entered as, for example, A-B-Space-Space-C (and vice-versa).

Numeric keys: All key comparisons and sequencing operations use string (not numeric) compares, so be careful if your keys are string conversions of numbers -- "10" sorts lower than "9". (For proper sorting, such keys, use fixed-length, right-justified, zero-filled numbers: "00009", "00010", etc.)

Forbidden Characters: You can't put any NULL (binary zero) or Control-A (binary one) characters into a key; **VsamEx** will detect them and tell your mother.

Compound Keys: If you like, you can think of each index as providing a "virtual sorting" of the records in a file. For instance, sequentially traversing a LastName index in an employee file is like reading a version of the file in which the records are sorted in last-name order. Just as traditional sort programs allow sorting on hierarchy of several "sort-keys" -- so that, for example, you could sort your employee records by date-of-hire within department -- you can create "compound" *index* keys that achieve the same effect in the virtual sorting of a **VsamEx** index.

To follow the above example, you'd design your records with an extra Field whose type would be “C5.3:6.10”. **VsamAddField** would construct a virtual field (it does not really exist in the record) and an index by concatenating department-name (real field 5 for length of 3 - on the left) and date-of-hire (real field 6 for a length of 10 - on the right). Internally, VsamEx will separated each concatenated field with a system delimiter and collapse multiple embedded spaces to a single space in its internal representations of the index. The specified lengths are only maximums and are used to limit the amount of data extracted from the beginning of the specified real fields to form the compound. If the string field contains less than the specified maximum, the smaller number of characters are still used in the compound. If no limit is specified, all of the data is taken up to the total key length of 252. *If the total length of the compound key (including separators) exceeds the 252 character maximum, an error will be returned.*

USING INDEXES

While a dataset is open, **VsamEx** maintains separate pointers into every index, including the primary index.

All record access takes place through the pointer into a selected index. When you want to read a record (which means using the **VsamGet** function), first decide which index to use.

When you're dealing with a given index, the pointers into the other indexes don't move. If you switch to a different index, **VsamEx** "remembers your place" in the index you had been using. Furthermore, we only use the pointers for reading, so you can add, change, and delete records without losing your place in the indexes.

You can visualize the indexes as separate Rolodex files, and the pointers as paper clips that mark your position (one paper clip per Rolodex). A pointer can be *on* an entry (attached to a specific card) or *between* entries, i.e., where a particular entry should be, but was not found.

There are also two special positions: before the first index entry (called "BOF," although really at the "Beginning of Index," not "Beginning of File"); or after the last entry (at "EOF"). **VsamEx** includes two special functions, **VsamBOF**, and **VsamEOF**, to move the pointer of a selected index to either of these two positions; also, when you first open a **VsamEx** dataset, all the pointers are set to BOF. Note that if an index is empty, BOF, and EOF are equivalent -- in a place only Zen masters can visualize.

Besides **VsamBOF** and **VsamEOF**, all pointer movement happens through use of the **VsamGet**, and the **VsamMovePtr** functions. There are two distinct phases to **VsamGet**: first, it moves the pointer in a selected index; second (and optionally), it reads the first record associated with the index entry under the just-moved pointer, only if that pointer movement ended up on (attached to) an entry. On the other hand, the **VsamMovePtr** function moves the pointer relative to the current position, either forward or backward by a specified number of records. This is accomplished at a very low level in VsamEx and is lightning fast.

VsamGet pointer movement, in turn happens in one of two distinct ways, depending on you specifications of an *access mode* parameter:

In **Lookup** access mode, **VsamGet** searches the index for an entry (key) that matches your *Selector* parameter. If it finds a match, it relocates the pointer to that entry. (This is the standard way to read a record "randomly" -- that is, by Key-match in a selected index.)

If it doesn't find a match, **VsamGet/**Lookup will still move the pointer -- *to where that entry would have been if it we there*; you can think of the pointer as coming to rest on a "phantom entry" that matches your *Selector* parameter. (This is called the "insertion point," since it's where a new entry with that value would be inserted.) This is an extremely useful process, as we'll see shortly.

The other three **VsamGet** access modes are "stepping" processes-that is, to move the pointer one "step" f*rom wherever it had been*. **VsamGet**/Next steps forward to the next entry; **VsamGet**/Previous steps backwards to the previous entry; and **VsamGet**/Current doesn't move the pointer at all so you can re-read the record.

Since all indexes are always kept sorted, you can read an entire file in any of several sort sequences by starting at BOF in an index and then looping on a call to **VsamGe**t/Next (in that same index). you can do the same thing in reverse sequence using EOF and Vsam**Get/**Previous.

If you're moving forwards in an index with a **VsamGet**/Next call, and try to do a read beyond the last entry, you'll get a "VIS_NOT_FOUND" return-code; you're at EOF, and subsequent **VsamGe**t/Previous call will read the last entry.

If you're moving backwards in an index with a **VsamGet**/Previous call, and try to read before the first entry, you'll get a " VIS_NOT_FOUND " return-code; you're at BOF, and a subsequent **VsamGe**t/Next call will read the first entry.

A " VIS_NOT_FOUND " return-code from a **VsamGet**/Current call tells you that you're either between entries, at BOF or at EOF.

To illustrate, let's suppose you have a file of three records, keyed with letters of the alphabet B, D, and E. The following demonstrates what happens when you do a record read (**VsamGet)** with various lookup keys. The Previous, Current, and Next columns show which record would be read (or that a not found condition would be returned) on a subsequent call -- after the read -- to **VsamGet/Previous**, **VsamGet/Next**, **VsamGet/Current**, respectively.

Lookup key	**Index pointer in file = ↑**	**Return Code LookUp**	**Results of a** *subsequent:* **Previous**	**Current**	**Next**
A	**B D E** ↑ *BOF* **before first entry**	**not found**	**not found**	**not found**	**B**
B	**B D E** ↑ **at B**	**OK**	**not found**	**B**	**D**
C	**B D E** ↑ **....between B and D**	**not found**	**B**	**not found**	**D**
D	**B D E** ↑ **.... .... at D**	**OK**	**B**	**D**	**E**
E	**B D E** ↑ **.... .... at E**	**OK**	**D**	**E**	**not found**
F	**B D E** ↑ **.... after last entry** *EOF*	**not found**	**E**	**not found**	**not found**

The Several **VsamGet** access modes are designed to be used together. In particular, this applies to Lookup and Next: Suppose you have a file of employee records with a LastName index, and you want to read all Smiths. This is a simple two-stage process: First, does a Lookup in the LastName index on "Smith." This will retrieve the first of potentially several Smiths (if it's successful; if it isn't, you don't have any Smiths, and you can stop). Second, *loop* on a call to **VsamGet**/Next in that same index, and keeps retrieving records until you go beyond the last Smith. (There's a coding illustration of this in the **VsamGet** function description.)

Now, recall the part about an "unsuccessful" **VsamGet**/Lookup, which doesn't find a match but relocates the pointer to a specific insertion point. Suppose you want to read all employee records with LastName beginning with the letters "Sm". Again, this is a two-stage process: First, do a Lookup in the LastName index on "Sm". You don't expect this call to report success (if it does, you have and employee whose last name is "Sm"). But it *will* position that pointer to the insertion point for "Sm". Then, loop on a call to **VsamGet**/Next in that name index, and keep retrieving records until you go beyond the last employee whose name begins with "Sm" (if there are any). (This is illustrated in the **VsamGet** function description.)

** There's a simple principle of alphabetic filing that's absolutely critical to understanding the last paragraph: Think of the insertion point for "Sm" as a "phantom" index entry. Such an entry would be filed immediately before all other entries that begin with the letters "Sm"--right? Now, please re-read the previous paragraph.*

VsamGet includes yet another parameter to help you with loop control in the Next and Previous modes, but that's secondary to the main subject.

Now, here are some interesting subtleties about pointer positioning, just in case you're getting bored. You can skip this part if you want, but it might come in handy if you're going to do certain kinds of fancy index maneuvering. It's all consistent, however, with the rest of the design. Trust me. (I'm a doctor.)

Suppose you do a **VsamGet**/Lookup, in a LastName index, for "Smith," and there isn't a Smith entry. As you know, the function will return "not found," and the pointer will be left positioned appropriately at the insertion point for Smith in the index, in the sequence where a Smith would be inserted. If you were to now do a **VsamGet/**Current call, it would, of course, also return "not found," since you're not on an entry. What may not be obvious is that this specific process -- an unsuccessful Lookup –leaves the pointer "primed" for that lookup key: if you now add the new entry Smith to the index, a subsequent **VsamGet**/Current call would reveal that the pointer had barked happily as soon as it saw the “Smith” record come in, and had wrapped its little paws tightly around it. Things are different once a pointer is actually attached to an index entry. No matter how the pointer landed on that entry -- by successful Lookup, by step-movement with either Next or Previous, or by the scenario described above -- once the pointer is attached, it gets "imprinted" to that entry, like a baby duck. (I know, ducks don't have paws but some programmers do!)

If, in our standard example, the LastName pointer were attached to the entry "Brown" and you then deleted the corresponding record - say with Social Security Number 111-22-3333 (our primary key in this example) - the pointer would again be primed; unlike the previous example, however, now the pointer would be primed for a *specific* entry; the Brown with primary key 11-22-3333, on which it had imprinted. If you re-add the *same* Brown back into the file, the LastName index pointer will indeed re-attach to his LastName entry -- but to no other Brown.

If you were instead, to add a Brown entry with a lower-sorting Social Security Number, it would be inserted above the pointer (closer to BOF, and Locatable with a **VsamGet**/Previous); if you were to add a Brown entry with a higher sorting Social Security Number, it would be inserted below the pointer (closer to EOF, and locatable with a **VsamGe**t/Next).

The Two different kinds of "pointer priming" -- one is indiscriminate, and the other specific -- actually make sense, and correspond to what your program logic would expect. In the First case, your Lookup intended to find the first of (potentially) several Smiths; the pointer ended up positioned after everything earlier than "Smith," and when a Smith came along, **VsamEx** gave it to you. (We aim to please.) In the second case, if you snatch a record away then immediately sneak it back in, things ought to look the same afterwards. In accordance with the Law of Universal Stupefaction, “Things that are not changed, are not changed!” In the interim, it wouldn't have made sense to arbitrarily reattach the pointer to a preceding or a following Brown. Were would you be then - to do a VsamGet/Current, you'd end up processing the wrong record.

While all of this might seem a bit tedious and confusing, it actually flows quite naturally from the Dictum of Defensive Design, which asks the familiar question: **"How else would you have done it?"**

Support Functions

VsamEx[treme] support functions are: (see the reference manual for a complete description)

VsamVal(lpDbName, lpLogBuf, Options, lpGcount, lpeKey)
VsamMakeMap(lpDbName, lpGcount, lpeKey)
VsamRebuild(lpDbName, lpLogBuf, Options, lpPhase, lpGcount, LpLicense, lpeKey)

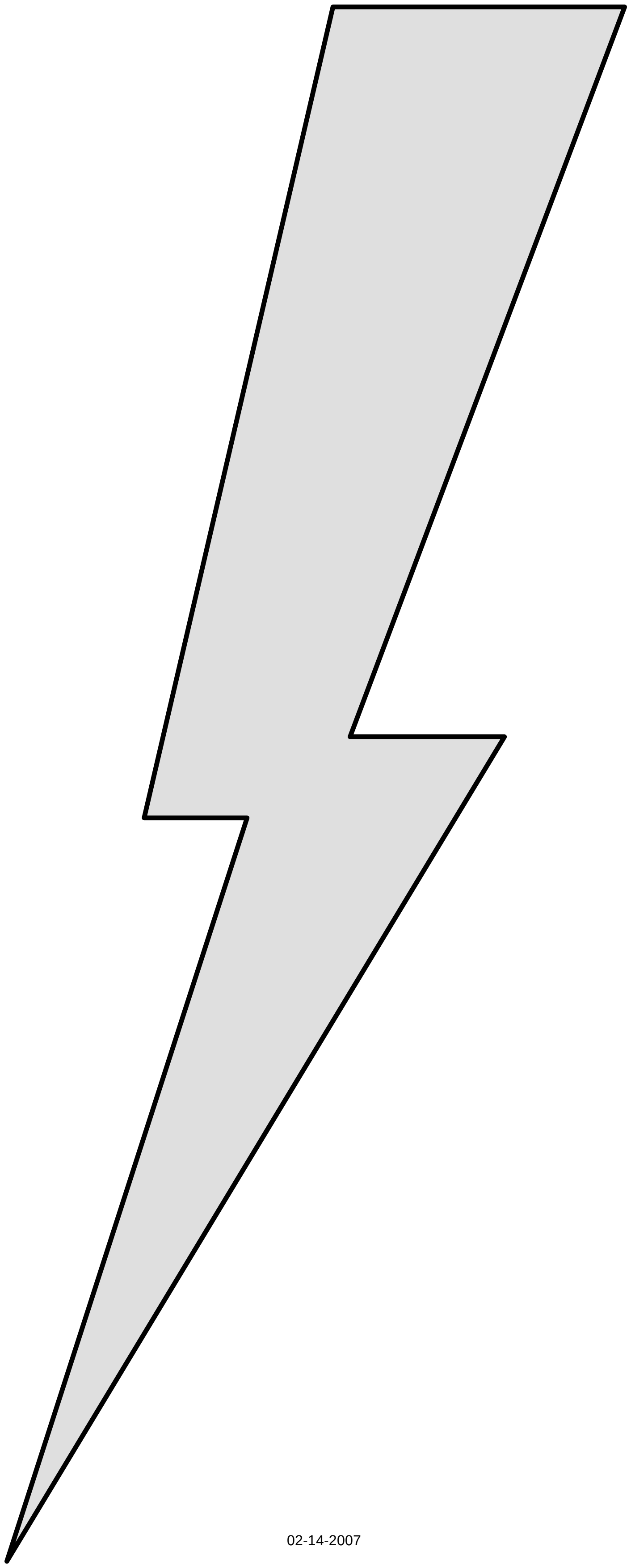
Once called, these functions will not return until complete. However:

In Windows they run with a message pump so other parts of the application will still function. The Gcount Variable, pointed to by the parameter lpGcount, is updated so other parts of the app can examine its progress.

In Linux, you should start another thread to run it if you want your current process to continue operating until the function call completes.

The DWORDs whose addresses are passed in parameters as lpPhase & lpGcount will continually change and are updated in real time. So, you may want to start the rebuild on its own thread. In Windows it runs with a message pump and threading is not mandatory. Remember, ***VsamRebuild*** will open the dataset in exclusive mode until it is complete. Once it starts however, it is complete only when the function returns to the calling process. If a secondary thread (or in a Windows Message Process or thread) calls the function again with **Options = -1**, this will signal the running process to stop! The running process will return immediately and at this point the rebuild process is incomplete! However, all of the original files in the dataset are untouched. ***VsamMakeMap,*** of course, is so fast that it does not end this way. The maximum number of reads it will perform is 65530 (one for each group).

NOTE*: We do not recommend running any of the support functions remotely. For performance reasons, they should only be run on the computer where the data resides. If your datasets are small to medium this is not as much of an issue.*



02-14-2007