Generic Callable Interface Programmer's Guide

Release 2.5 of the Callable Interface

Publication Number
007-6319-001

November 19, 2013

Pubication Number 007-6319-001

Contents

Welcome xiii

Purpose of This Document xiii

Intended Audience xiii

Contents xiii

Related Documentation xiv

Notational Conventions xv

Chapter 1: Introduction 1-1

C Language Interface 1-1

Callable Interface Function Hierarchy 1-1
- Session Control Functions 1-2
- File Operation Functions 1-2
- Data Transfer Functions 1-3
- StorHouse Command Functions 1-3

Chapter 2: StorHouse Parameters and Data Descriptions 2-1

Session and Data Transfer Link Identifiers 2-1

StorHouse Accounts and Passwords 2-1
- Account Passwords 2-2
- Default Access Groups and Rights 2-2
- StorHouse Privileges 2-2

StorHouse Files and File Access Groups 2-3
- StorHouse File Names 2-3
- File Access Groups 2-4

Group and File Access Passwords 2-4
Group Passwords 2-4
File Passwords 2-4
File Versions 2-5
File Revisions 2-6
File Data Representation 2-6
Directory Information 2-6

Chapter 3: File Positioning 3-1

Record Sequencing 3-1
Entry Sequence 3-1
Key Sequence 3-1
Current Record Position 3-2
Read Functions and Current Record Position 3-2
Record Sequencing Example 3-2

Chapter 4: Control Structures for the C Interface 4-1

Parameter Values 4-1
Character Strings 4-1
Externally Specified Parameters 4-2
Tokens 4-2
Return Codes 4-2
Indicative Text Messages 4-3

Chapter 5: Definition of the LSMxxx Functions 5-1

Synchronous and Asynchronous Functions 5-1
Synchronous Mode 5-1
Asynchronous Mode 5-1
Function Statement Format 5-2
Session Control Functions 5-2
LSMCON - Connect 5-3
Function Prototype Definition 5-3
Argument Description 5-3
Return Codes 5-4
Detailed Function Description 5-4
Cross-Reference to Sample C Program 5-4
LSMDIS - Disconnect 5-5

Function Prototype Definition 5-5
Argument Description 5-5
Return Codes 5-5
Detailed Function Description 5-5
Notes 5-5
Cross-Reference to Sample C Program 5-5

File Operation Functions 5-6

LSMOS - Open Sequential 5-7
Function Prototype Definition 5-7
Data Structures 5-7
Argument Description 5-8
Return Codes 5-12
Detailed Function Description 5-12
Notes 5-13
Cross-Reference to Sample Program 5-13

LSMCO - Create Open 5-14
Function Prototype Definition 5-14
Data Structures 5-14
Argument Description 5-15
Return Codes 5-18
Function Description 5-19
Notes 5-19
Cross-Reference to Sample Program 5-19

LSMOV - Open VRAM 5-20
Function Prototype Definition 5-20
Data Structures 5-20
Argument Description 5-21
Return Codes 5-23
Detailed Function Description 5-23
Notes 5-24
Cross-Reference to Sample Program 5-25

LSMCP – Checkpoint 5-26
Function Prototype Definition 5-26
Argument Description 5-26
Return Codes 5-26
Detailed Function Description 5-26
Notes 5-27
Cross-Reference to Sample Program 5-27

LSMCLO – Close 5-28
Function Prototype Definition 5-28
Argument Description 5-28
Return Codes 5-28
Detailed Function Description 5-29

Notes 5-29
Cross-Reference to Sample Program 5-29

Data Transfer Control Functions 5-30

LSMR – Read 5-31
Function Prototype Definition 5-31
Argument Description 5-31
Return Codes 5-31
Detailed Function Description 5-31
Note 5-32
Cross-Reference to Sample Program 5-32

LSMRS – Read Sequential 5-33
Function Prototype Definition 5-33
Argument Description 5-33
Return Codes 5-33
Detailed Function Description 5-34
Notes 5-34
Cross-Reference to Sample Program 5-34

LSMRR – Read Record 5-35
Function Prototype Definition 5-35
Argument Description 5-35
Return Codes 5-35
Detailed Function Description 5-36
Notes 5-36
Cross-Reference to Sample Program 5-36

LSMRK – Read Keyed 5-37
Function Prototype Definition 5-37
Argument Description 5-37
Return Codes 5-38
Detailed Function Description 5-38
Notes 5-38
Cross-Reference to Sample Program 5-38

LSMRNK – Read Next Key 5-39
Function Prototype Definition 5-39
Argument Description 5-39
Return Codes 5-39
Detailed Function Description 5-40
Notes 5-40
Cross-Reference to Sample Program 5-40

LSMW – Write 5-41
Function Prototype Definition 5-41
Argument Description 5-41
Return Codes 5-41

Detailed Function Description 5-41
Notes 5-42
Cross-Reference to Sample Program 5-42

LSMWK – Write Key 5-43
Function Prototype Definition 5-43
Argument Description 5-43
Return Codes 5-43
Detailed Function Description 5-44
Notes 5-44
Cross-Reference to Sample Program 5-45

LSMDEL – Delete Record 5-46
Function Prototype Definition 5-46
Argument Description 5-46
Return Codes 5-46
Detailed Function Description 5-46
Note 5-46
Cross-Reference to Sample Program 5-46

LSMCH – Change Record 5-47
Function Prototype Definition 5-47
Argument Description 5-47
Return Codes 5-47
Detailed Function Description 5-47
Note 5-47
Cross-Reference to Sample Program 5-48

StorHouse Command Submission 5-49

LSMSCI – StorHouse Command Interface 5-50
Function Prototype Definition 5-50
Data Structures 5-50
Argument Description 5-50
Return Codes 5-51
Detailed Function Description 5-52
Notes 5-53
Cross-Reference to Sample Program 5-53

General Usage Functions 5-54

LSMCK – Check 5-55
Function Prototype Definition 5-55
Argument Description 5-55
Return Codes 5-55
Detailed Function Description 5-55
Notes 5-55
Cross-Reference to Sample Program 5-56

LSMASY – Set Asynchronous Mode 5-57

Function Prototype Definition 5-57
Argument Description 5-57
Return Codes 5-57
Detailed Function Description 5-57
Cross-Reference to Sample Program 5-58

LSMMSG – Message Retrieve 5-59
Function Prototype Definition 5-59
Argument Description 5-59
Return Codes 5-59
Detailed Function Description 5-59
Notes 5-60
Cross-Reference to Sample Program 5-60

LSMAB – Abort 5-61
Function Prototype Definition 5-61
Argument Description 5-61
Return Codes 5-61
Detailed Function Description 5-61
Notes 5-62
Cross-Reference to Sample Program 5-62

Chapter 6: Sample Program 6-1

LSMxxx Sample Program 6-2

Appendix A: Checkpoint/Restart and Programming Guidelines A-1

Checkpoint/Restart A-1
Examples A-1
Example 1 .... A-2
Example 2 .... A-2
Example 3 .... A-3
Example 4 .... A-3

Programming Guidelines A-4
Defining Resources A-4
Examples A-5
User Guidelines A-6
Permanent Fixes A-6

Appendix B: CREATE FILE Command B-1

CREATE FILE B-1
Format B-1

Description B-2
Parameters B-3
Command Modifier B-3
Parameter Modifiers B-3

Estimating a Value for /SIZE B-12
Estimating Data Extent Size B-12
Estimating DF Extent Size B-13
All Files B-13
Checkpointed Files B-13
Files with Keys B-13
Keysequential Files B-14
Updated Files B-14
Estimating K Extent Size B-14
Example 1 ..... B-15
Example 2 ..... B-17

Key-Definition Mode B-19
KEY Command B-19
Defining Keys B-20
Concatenating Keys B-21
Commenting KEY Commands B-21
Sample Key Definition Series B-22
EXIT Command B-22

Examples B-22

Appendix C: Installing and Using the StorHouse API for Windows C-1

Installing the Windows StorHouse API C-1
Installation Requirements C-1
Installation Diskette Contents C-2

Compiling and Linking the Sample Code With the StorHouse API for Windows C-2
Sample Programs C-2
FILETEK.INI File C-3
Using DLL C-3

Appendix D: Installing and Using the OS/2 StorHouse API D-1

Installing the OS/2 StorHouse API D-1
Installation Requirements D-1
Installation Disk Contents D-1
Installation Steps D-2

Compiling and Linking an Application With the OS/2 StorHouse API D-2
Microsoft C 6.0 Compiler and Linker Switches D-3

Linking the OS/2 StorHouse API D-3
Compiling and Linking Method D-3
Sample Makefile D-4

Index Index-1

Welcome

The *Generic Callable Interface Programmer's Guide* describes the StorHouse® Callable Interface for all hosts *other than* IBM™ MVS™ hosts. This interface provides access to StorHouse from end-user applications. For information on the StorHouse Callable Interface for IBM MVS hosts, please refer to the *Callable Interface Programmer's Guide* (publication number 900013).

Purpose of This Document

This document is a reference manual that describes the functions in the StorHouse Generic Callable Interface. It presents a prototype definition, an argument description, return codes, a detailed functional description, and a cross-reference to the sample program in Chapter 6 for each function.

Intended Audience

The *Generic Callable Interface Programmer's Guide* was written for programmers who write applications that invoke the Callable Interface.

Contents

This manual contains six chapters and four appendices.

- Chapter 1, “Introduction,” briefly describes the Callable Interface.
- Chapter 2, “StorHouse Parameters and Data Descriptions,” provides basic information about StorHouse data items that are required arguments to the

Callable Interface functions. These items include user identification, file identification, and security information.

- Chapter 3, “File Positioning,” discusses record sequencing and the effect of different read functions on file position.
- Chapter 4, “Control Structures for the C Interface,” discusses the general format of data areas that are used in many of the LSMxxx functions.
- Chapter 5, “Definition of the LSMxxx Functions,” contains a detailed definition of all Callable Interface functions for the C language.
- Chapter 6, “Sample Program,” provides a sample C program.
- Appendix A, “Checkpoint/Restart and Programming Guidelines,” contains information about using the checkpoint/restart capability and programming guidelines.
- Appendix B, “CREATE FILE Command,” describes the StorHouse Command Language CREATE FILE command.
- Appendix C, “Installing and Using the StorHouse API for Windows,” contains information about installing and using the Windows StorHouse API.
- Appendix D, “Installing and Using the OS/2 StorHouse API,” contains information about installing and using the OS/2 StorHouse API.

Related Documentation

Programmers who use the Callable Interface should be familiar with the following StorHouse documents found in the StorHouse User Document Set:

- The *Messages and Codes Manual*, publication number 900032, describes the messages and return codes generated by the StorHouse system and host software.
- The *Command Language Reference Manual*, publication number 900005, explains StorHouse Command Language in detail.
- The *StorHouse Glossary*, publication number 900027, defines terminology used in SGI publications.

Notational Conventions

This book uses the following conventions for illustrating command formats, presenting examples, and identifying special terms:

Convention	Meaning
Angle brackets (< >)	Enclose optional entries
Braces ({ })	Enclose descriptive terms or a choice of entries
Courier font	Code
Ellipses (...)	A repetition of the preceding material
Italics	New terms and emphasized text
lower case Helvetica font	User entries
UPPER CASE	System responses and StorHouse terms

Chapter
1

Introduction

The StorHouse Callable Interface provides access to StorHouse from user application programs. The Interface is a collection of callable subroutines that allow the application to perform file maintenance and status operations, and data transfers to and from StorHouse.

C Language Interface

The C language interface (also usable with C++ compilers) is a set of functions named LSMxxx, where xxx varies with the specific StorHouse operation. Designed for portable programs, this interface is available for most machine and operating system environments.

Callable Interface Function Hierarchy

Figure 1-1 illustrates how Callable Interface functions are organized in a hierarchical structure.

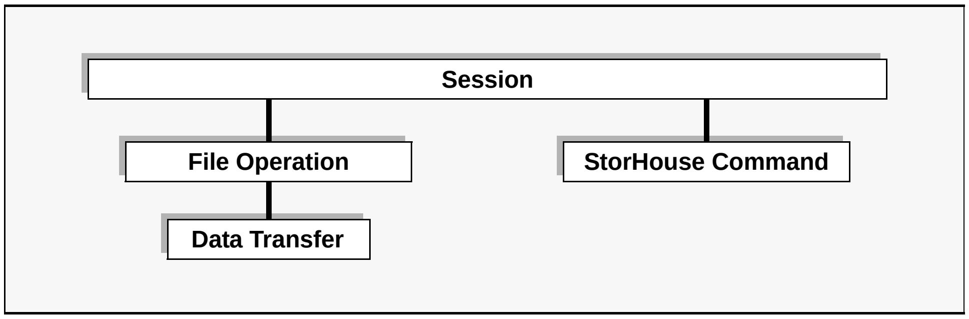

Figure 1-1: Callable Interface Function Hierarchy

To perform any StorHouse operation, an application must:

1. Establish a session with StorHouse using the session function LSMCON.
2. Within a session, initiate file operations using one of the open functions.
3. Once a file is opened, issue data transfer operations using one of the read/write functions; issue record update operations using the change or delete function; or issue StorHouse maintenance/status commands.

In addition, with an established session, a program can issue StorHouse commands such as SHOW FILE or SHOW ACCOUNT.

Session control, file operation, data transfer, and StorHouse command functions are described in the following sections.

Session Control Functions

There are two Callable Interface session control functions: the connect function LSMCON and the disconnect function LSMDIS.

- LSMCON establishes a session between a user application and StorHouse. The application must supply an account identifier and password for the connect function so that the StorHouse security system can validate the session and set session privileges and defaults.

- LSMDIS ends the established session and releases all session-related host and StorHouse resources.

One application can establish several sessions. StorHouse assigns each session a unique session identifier to allow explicit application control of the operations that are performed in that session. LSMCON returns the session identifier, the c_token, to the application.

File Operation Functions

File operation functions initiate and terminate file operations. These functions provide a data transfer path within the session. There are five file operation functions.

- LSMOS opens sequential files and allows file-oriented operations that read or create a complete StorHouse file.

- LSMCO creates a new VRAM™ file on StorHouse and then establishes a data transfer link for writing data to that file.

- LSMOV allows read, write, delete, and update access to individual records in StorHouse VRAM files.

- LSMCP synchronizes file transfer by ensuring that all previously written records have been received and processed by StorHouse.
- LSMCLO terminates the file operation (indicating end-of-file for write operations) and releases all resources used by the transfer operation.

The open functions establish a data transfer link between the user application and StorHouse. These functions return the o_token, which identifies the data transfer path within a given session. The o_token also identifies the file being processed in all subsequent transfer-oriented functions (for example, read and write).

Data Transfer Functions

Data transfer functions operate in an established data transfer path on specific records in a file. Sequential read and write functions move the next record from StorHouse to the host and to StorHouse from the host, respectively. If VRAM file organization is used, individual records can be retrieved by record number or record key with change-record and delete-record capabilities.

StorHouse Command Functions

StorHouse command functions allow an application to send Command Language commands to StorHouse and to retrieve response text from those commands. These functions also allow administrative operations to be directed from an application rather than from a user at a terminal through the Interactive Interface.

Chapter

2

StorHouse Parameters and Data Descriptions

This chapter contains general information about StorHouse data transfer link identifiers, accounts, file access groups, and files. For more information about these topics, refer to the *StorHouse Concepts and Facilities Manual* and the *Command Language Reference Manual.*

Session and Data Transfer Link Identifiers

There are two types of link identifiers: session and data transfer.

- *Session identifier* – LSMCON returns a c_token, the session link identifier.
- *Data transfer link identifier* – LSMOS, LSMCO, and LSMOV return an o_token, the data transfer link identifier.

The c_token and o_token must be passed to all other Callable Interface functions that perform operations with the session or data link. Token values can be moved from one variable to another, but they must not be subjected to any arithmetic operations, including type conversions.

StorHouse Accounts and Passwords

An *account* is a collection of administrative data that StorHouse uses to control a session. Each account includes an *account identification code* (AID) and a password. The AID is similar to the user name. It provides StorHouse with the user’s identity. An AID must contain 1 to 12 characters and include only the following characters: A-Z, 0-9, $, and _ (underscore).

A program must include an account identifier to establish a session in the Callable Interface. Multiple programs can use the same or different accounts and can access StorHouse at the same time.

Account Passwords

Account passwords help maintain system security. Passwords must contain 0 (null) to 32 characters and include only the following characters: A-Z, 0-9, $, or _ (underscore). Generally, long passwords provide better system protection than short passwords. Passwords of three or fewer characters offer only marginal protection.

Default Access Groups and Rights

Generally, each account has a *default access group* as well as *default access rights* to that group. When a command accesses a file, StorHouse assumes the file is in the default group unless a different group is specified in the command.

An account may be set up to give read, write, delete, or no default access to the default group. These are the access rights to the default group. Thus, a program does not have to give a group access password to perform operations on files in the default group if the account's default access rights include the type of access required to perform the operation. For example, if an operation requires write access and the account has write access to the group by default, the write password need not be specified.

In any case, if the default group has a null password, a program automatically receives the corresponding type of access without having default access or specifying a password. A program can switch to a different default access group during a session.

For information about specifying account information, refer to the section entitled "Session Control Functions" on page 5-2 and to Chapter 6, "Sample Program."

StorHouse Privileges

Each account has a set of privileges. Privileges fall into two categories:

- *Access privileges* – allow the program using the account to bypass various system security checks.
- *Command privileges* – permit the program using the account to perform specific commands or groups of commands.

The privileges assigned to an account determine the functions that an application program can perform.

For a complete list of access and command privileges, see the *Command Language Reference Manual*.

StorHouse Files and File Access Groups

A *file* is a named collection of logically related data treated as a unit by StorHouse and located on a medium – for example, magnetic or optical disk. Any collection of data generated by a host program can be stored in StorHouse. Each file has a set of attributes that govern where and how it is stored in StorHouse.

Each file can be protected by passwords. A file can be locked to prevent programs using other accounts from reading or writing to it, and unlocked to make it available to other programs using other accounts.

StorHouse File Names

StorHouse uses file names to identify files. A *file name* is a unique name that identifies a file within an access group. File names must contain 1 to 56 characters and at least one character must be non-blank. Uppercase characters are distinct from lowercase characters.

StorHouse file names must consist of printable ASCII characters, as shown in Table 2-1.

Table 2-1: Characters Allowed In StorHouse File Names

PRINTABLE ASCII CHARACTERS					
A-Z	uppercase letters	+	plus sign	()	parentheses
a-z	lowercase letters	~	tilde	< >	angle brackets
0-9	digits	,	comma	[ ]	square brackets
!	exclamation point	-	hyphen	{ }	braces
“	quote	.	period	\	backslash
#	number sign	/	slash	^	circumflex
$	dollar sign	:	colon	_	underscore
%	percent sign	;	semicolon	|	vertical bar
&	ampersand	=	equal sign	‘	reverse apostrophe
’	apostrophe	?	question mark		space
*	asterisk	@	at sign		

File Access Groups

A *file access group* is a set of files. Program access may be restricted to a file access group. To manipulate files in the set, a program must specify a group name and, if the group is protected by passwords, a read password, a write password, and/or a delete password. Each file in the StorHouse system is a member of one group. Within that group, each file name is unique; however, two files may have the same name if they are located in different file access groups. Group names must contain 1 to 8 characters and include only the following characters: A-Z, 0-9, $, or _ (underscore).

Group and File Access Passwords

StorHouse allows the specification of group and file passwords to protect user files from unauthorized access. The following sections explain how to use group and file passwords.

Group Passwords

Group passwords protect user files from unauthorized access. Passwords may be null or contain 1 to 8 characters, and include the following characters: A-Z, 0-9, $, and _ (underscore).

Group passwords are used as follows:

- If a group has a read password, a program must specify the correct read password to read a file from StorHouse to display group or file directory information.
- If a group has a write password, a program must specify the correct write password to write a file into the group or to UNDELETE a file.
- If a group has a delete password, a program must specify the correct delete password to delete the group, delete a file from the group, change the group's passwords, or change passwords or file attributes in the group.
- If a group has a null password, a program may gain the corresponding type of access by specifying a null password or by not specifying a password.
- If a program specifies a null password where the group has a non-null password, StorHouse does not grant that type of access (in other words, read, write, or delete access). However, StorHouse returns an error if that type of access is required.

File Passwords

A program can also give individual files read, write, and delete passwords. These file passwords control access to files in the same way that group passwords control access to file access groups.

File passwords can be null or contain 1 to 8 characters. Like group passwords, file passwords may contain only the following characters: A-Z, 0-9, $, or _ (underscore).

File passwords are used as follows:

- If a file has a read password, the program must specify the correct read password to read the file from StorHouse or display directory information about the file.
- If a file has a write password, the program must give the write password to write a new version of the file into StorHouse or to UNDELETE a file.
- If a file has a delete password, the program must specify the correct delete password to delete the file from StorHouse, or change the file's attributes or passwords.
- If a file has a null password, the program may gain the corresponding type of access by specifying a null password or by not specifying a password.

The account used by the program must have read, write, or delete access to a file's group before the system allows the program to gain the corresponding access to the file.

If a program specifies a null password where the file has a non-null password, StorHouse does not grant that type of access (in other words, read, write, or delete access). However, StorHouse returns an error if that type of access is required.

For more information about specifying file access group names and group and file access passwords, refer to "Data Transfer Control Functions" on page 5-30 and to Chapter 6, "Sample Program."

File Versions

A new *file version* is created whenever a program transfers a non-VRAM file from the host to StorHouse. A new version of a VRAM file is created whenever a CREATE FILE command is performed or a LSMCO (Create Open) call is issued with a checkpoint of 0. The new file receives version number 0. If a file of the same name and the same access group already exists in StorHouse, the number of each previous version decreases by one. Previous versions may range in number from -1 through -32767 or from -1 through the minimum version number allowed by the file's LIMIT attribute. If there was a previous version -32767, it is deleted when a new version is added to the StorHouse system.

For example, when the file DATAFILE is first added to StorHouse, it becomes version 0. When a new version of DATAFILE is added, it becomes version 0, and the previous version 0 of DATAFILE becomes version -1.

Refer to the description of LSMOS on page 5-7 or LSMOV on page 5-20 for information about specifying version numbers.

File Revisions

For RECORD, KEYED, or KEYSEQUENTIAL VRAM files, StorHouse assigns revision number 1 to a file version when the file is created on the StorHouse system. Each time a user changes the contents of the file version, StorHouse increments the revision number by 1. A user can change the contents of a file version by opening the file; by changing, deleting, or adding records; and by closing the file. Thus, a file version can have multiple revisions, each identified by a unique revision number.

Revision numbers can be expressed as relative or absolute numbers. Relative revision numbers range from 0, the current revision, through -65534, the oldest revision. Absolute revision numbers range from 1 through 65535. For example, assume that relative version 0 of the file DATAFILE has four revisions. A user can refer to the most current revision of this file as relative revision number 0 or as absolute revision number 4.

Refer to the description of LSMOV on page 5-20 for more information about specifying values for revision numbers.

File Data Representation

StorHouse stores the record stream written by the user application program either as binary (bitstream) data or as an ASCII character stream. The file format is determined when the file is created, either by the LSMOS function for sequential files, or by the CREATE FILE command or the LSMCO (Create Open) function for VRAM files. (For information about the CREATE FILE command, refer to Appendix B, “CREATE FILE Command.”) The record data is treated as a bitstream unless the data translate flag in the file attributes array is positive in a LSMOS function or the /ASCII modifier is specified on the CREATE FILE command.

Files created through the Callable Interface are considered transportable by StorHouse. They can be accessed by host computers from different manufacturers running different operating systems. For binary records, the user application program is responsible for any required data conversion. For ASCII files, the host translates from ASCII to the host character code.

Directory Information

The StorHouse directory entry for a file indicates whether the file’s record format is binary bitstream or ASCII character stream by the value of the file system type. File system type is set to 65 for ASCII files and 66 for binary files. Files created by SGI host file utility programs are given other file system type identifiers, based on the specific utility used to copy the file.

Chapter 3

File Positioning

This chapter discusses record resequencing and explains how the various functions affect record positioning for files.

Record Sequencing

Records in a keyed VRAM file are sequenced by both entry and key. Entry sequenced records are sequenced by the order in which they were written to the file. Key sequenced records are sequenced by the values of key fields in each record.

Entry Sequence

The write operation that builds a file determines the entry sequence for records in that file. Each new record is appended to the end of the file, independent of record content. Entry sequence determines the order in which sequential read operations retrieve records. Entry sequence has no effect on the order in which key value read operations and next-key sequence read operations retrieve records.

Key Sequence

Each key that is defined for a file determines a key sequence for records in that file. In key sequence, records are ordered by the increasing value of their key field, which is considered only as a binary bitstring. Key sequence determines the order in which *next-key* read operations retrieve records. The order in which records with duplicate keys are returned is not necessarily the same as their entry sequence.

Current Record Position

Any opened StorHouse file has a *current* record position. A keyed VRAM file has two current record positions:

- *Sequential* position, which follows record entry sequence.
- *Key* position, which follows key sequence.

A file can have several keys but only one key position. Key position is always relative to the last key that was used to read a record.

For a keyed VRAM file, open sets the sequential record position to the beginning of the file, which is the first record in entry sequence. Open does not initialize key position.

Read Functions and Current Record Position

Four functions can be used to read a VRAM file:

- LSMRS – retrieves the next record in entry sequence order.
- LSMRK – retrieves a record by exact match of a specified value for a given key.
- LSMRNK – retrieves the next record in key sequence order.
- LSMRR – retrieves a specific record by record number.

Read functions maintain current record position for a file as follows:

- Every read operation updates sequential record position.
- Only LSMRK and LSMRNK update key position. LSMRK sets the current key position and must be called at least once prior to calling LSMRNK.

Record Sequencing Example

The following example shows a VRAM file with two keys: NAME and ENUM. The file was created by writing the following records, where record number matches the entry sequence.

Record Number	Value of Key NAME	Value of Key ENUM
1	Jones	327
2	Smith	409
3	Doe	427
4	Johnson	283
5	Smith	265
6	Brooks	301

If the file is opened and read sequentially, then the records are read in the following order: 1, 2, 3, 4, 5, 6.

The following table indicates how each read function affects the file's current record position.

Read Operation Function			Record No.	NAME Key	ENUM Key
1.	LSMOV		None		
2.	LSMRS		1	Jones	327
3.	LSMRR		4	Johnson	283
4.	LSMRS		5	Smith	265
5.	LSMRK	(Key=NAME (Value=DOE)	3	Doe	427
6.	LSMRNK		4	Johnson	283
7.	LSMRS		5	Smith	265
8.	LSMRK	(Key=ENUM) (Value=283)	4	Johnson	283
9.	LSMRNK		6	Brooks	301
10.	LSMRK	(Key=NAME) (Value=SMITH)	2	Smith	409
11.	LSMRNK		5	Smith	265
12.	LSMRS		6	Brooks	301

Note that in the preceding table, operations 10 and 11 may return record number 5, then record number 2. The order of duplicate key records may be changed by the file's update and delete history.

Chapter

4

Control Structures for the C Interface

The include file LSMDEFS is supplied with the C Interface. LSMDEFS contains function prototype and structure definitions for all C Interface functions (LSMxxx).

Parameter Values

The following sections describe how to specify values for parameters.

Character Strings

The character set for a string-valued parameter consists of the uppercase letters (A-Z), the lowercase letters (a-z), the digits 0-9, and the following special characters:

```
, : “ ‘ ! - ( ) . * /
```

A character string must be supplied as a pointer to a standard C null-terminated string (“char*”). The documentation for each string-valued parameter specifies whether the value is restricted to a subset of this character set or the value is case-sensitive. Values that are not case-sensitive can be supplied in either upper or lower case. The documentation also specifies the maximum length of each string-valued parameter. That length does not include the null terminator. The include file LSMDEFS contains definitions for all field length maximums. The parameter documentation also indicates the defined name for each field length maximum.

Note that a key value is not a string-valued parameter; it is a variable-length array of 8-bit bytes.

Externally Specified Parameters

The values for some string-valued parameters can be externally supplied through symbolic variables. (The mechanism is operating system dependent.) Where applicable, these parameters are documented for each function. When the string value specified by the program begins with the characters DD=, the characters following the equal sign are used as a symbolic variable, which is translated (via operating system dependent function calls). The result of the translation is used as the actual parameter value.

Tokens

The session connection function and the file transfer open functions set up an object, or *token*, to identify the session or transfer link, respectively.

- The *connect* token (c_token) is built by the session connection function.
- The *open* token (o_token) is built by the data transfer open functions.

Tokens are implemented as structures. The caller must supply memory for the token structure and must pass a pointer to that structure as the first parameter to all LSMxxx function calls.

The token structure is defined in the include file LSMDEFS as structure LSMS_TOKEN. The size of this structure is three pointers (of type “char*”).

The caller does not need to initialize this structure prior to calling LSMCON, LSMOS, LSMCO, or LSMOV. The application program should not directly reference or change the members of this structure. This is accomplished with the LSMASY function.

Return Codes

Return codes are of type int and are binary integer values between 0 and 8191. The return code 0 always indicates normal completion. Common return codes for each function are documented in the function description sections in Chapter 5, “Definition of the LSMxxx Functions.”

Note In 16-bit and 32-bit environments, all function prototypes for the LSMxxx functions return a `short` integer (2 bytes in length) rather than `int`. For more information, see Appendix C, “Installing and Using the StorHouse API for Windows.”

All return code values are documented in the *Messages and Codes Manual*.

Indicative Text Messages

A Callable Interface function may generate text messages that provide commentary, warnings, or error diagnostics associated with the processing of the function. These messages are text strings that can be printed or displayed at a terminal.

These messages are not returned directly by the function. They are placed in a message stack and can be retrieved only by calling LSMMSG, the message retrieve function. Users may ignore these messages; the stack is cleared when the next function request is made.

The indicative text message stack is normally cleared when the session or the data transfer operation is ended. However, clearing the text message stack in this manner also deletes any messages that were generated during the session disconnect or during the transfer close operation. It is the user's responsibility to indicate whether these messages are retrieved, either when the session is established or when the data transfer is opened. The definition sections for LSMCON (Connect), LSMOS (Open Sequential), LSMCO (Create Open), and LSMOV (Open VRAM) document the use of a flag to control this message retention capability.

A non-zero return code does not guarantee that an indicative message is available. Conversely, a zero return code does not guarantee that there are no messages in the stack.

Chapter

5

Definition of the LSMxxx Functions

Chapter 5 contains a detailed definition of all Callable Interface functions for the C language. All Callable Interface capabilities for the C language are invoked through a set of functions named LSMxxx. LSMxxx functions may operate in synchronous or asynchronous mode.

Synchronous and Asynchronous Functions

This Callable Interface implementation does not provide truly asynchronous processing. Asynchronous mode operation and the LSMCK function are provided only to allow for future compatibility.

Synchronous Mode

In synchronous mode, control is returned to the user program only after all processing associated with the requested function has been completed. For some functions, this means that the request has been passed to StorHouse and that StorHouse has completed all processing associated with that request. For other functions, completion means only that the user program can continue as though all processing associated with the function has been completed, although the request may only have been queued for subsequent StorHouse processing. For example, LSMW (Write) signals completion when data has been moved from the user buffer to an internal buffer.

Asynchronous Mode

In asynchronous mode, control is returned to the user program as soon as the request has been queued. The user must call the LSMCK (Check) function prior to using the results of the request.

All function calls are synchronous unless the application program explicitly calls LSMASY to set the token to the asynchronous operation state. All tokens are set to the synchronous processing state when they are initialized (by LSMCON, LSMOS, LSMCO, or LSMOV). Once set for asynchronous processing by LSMASY, all subsequent function calls using that token operate in asynchronous mode. LSMASY also provides the capability to reset the mode to synchronous.

Function Statement Format

The LSMxxx functions are grouped in the following categories:

- Session Control
- File Operations
- Data Transfer Control
- StorHouse Command Submission
- General Usage.

The following sections describe these functions.

Session Control Functions

Two session control functions allow an application to begin or end a StorHouse session. These functions are:

- LSMCON
- LSMDIS.

The following sections describe LSMCON and LSMDIS.

LSMCON - Connect

LSMCON establishes a session with StorHouse. A session must exist before any other functions can be called.

Note This function is slightly different for Microsoft® Windows™ environments. For more information, see Appendix C, “Installing and Using the StorHouse API for Windows.”

Function Prototype Definition

```
extern int LSMCON  ( struct LSMS_TOKEN *c_token,
                     int message_flag,
                     char *account,
                     char *password,
                     char *sm_identifier,
                     char *subsystem_identifier
                   );
```

Argument Description

c_token	A pointer to a token (LSMS_TOKEN) structure. The structure need not be initialized; it is filled in by the LSMCON function to contain session identification and to clear the asynchronous processing indicator. The application program should not manipulate the members of this structure. It should only be used as the connect token (c_token parameter) to other LSMxxx calls for this session.
message_flag	An integer that indicates how indicative text messages are handled. If non-zero (defined value LSMF_MSGHOLD), then text messages from all session errors, including connect/disconnect function errors, are retained in the message stack. If zero (defined value LSMF_NOMSGHOLD), messages may not be retrievable after the session has terminated.
account	A character string containing the StorHouse account identification code (AID) for the session. This field allows only a restricted character set. Lowercase characters may be specified but are treated as uppercase characters. The only special characters allowed are _ (underscore) and $. The maximum length for this string is 12 (defined value LSML_AIC).
password	A character string containing the password associated with the StorHouse account. The character set has the same restrictions as those for the account parameter. The maximum length for this string is 32 (defined value LSML_SOPW).

sm_identifier	A character string that identifies the specific StorHouse system to be accessed. If null, the default or only StorHouse system is accessed. The maximum length for this string is 6 (defined value LSML_SMID). This argument represents the sm_name entry in the SMCONFIG file. For more information on the SMCONFIG file, see the *Host Software Installation Guide for UNIX Hosts.*
subsystem_identifier	A character string with a maximum length of 4. This argument is not used and should be null.

Return Codes

Any Non-Zero Value	Indicates that the session was not established. In this case, the LSMDIS (Disconnect) function should not be called. If message_flag was set (non-zero), then the LSMMSG (Message) function must be called until all messages have been retrieved.

Detailed Function Description

The first step in any interaction with StorHouse is to establish a session. This is accomplished by calling LSMCON and by supplying the account identification code and security information required by StorHouse. After the session is established, other Callable Interface functions may be called.

If message_flag is set (non-zero), then LSMMSG must be called after the session ends. The dynamic memory allocated for the session is not released until all messages have been retrieved; that is, until a return code of 3065, indicating no more messages, is received from LSMMSG.

Cross-Reference to Sample C Program

See Step 1 in the sample program in Chapter 6, “Sample Program.”

LSMDIS - Disconnect

LSMDIS concludes a StorHouse session by terminating the connection that was established by LSMCON.

Function Prototype Definition

```
extern int LSMDIS ( struct LSMS_TOKEN *c_token);
```

Argument Description

c_token	A pointer to a session identifier token (connect token).

Return Codes

Any Non-Zero Value	Indicates that the session was not concluded successfully and that resources allocated by the Callable Interface routines may not be released.

Detailed Function Description

The final step in any interaction with StorHouse is to terminate the session. This is accomplished by calling LSMDIS. The session is identified by the c_token returned by LSMCON. A successful (return code zero) disconnect terminates the session and releases all resources allocated by the StorHouse Callable Interface functions.

Notes

- File operations should be explicitly closed before signing off. Otherwise, LSMDIS terminates the data transfer operation with an abort status.
- If the message flag was set when the session was established (see message_flag under LSMCON on page 5-3), then LSMMSG should be called following the LSMDIS call.

Cross-Reference to Sample C Program

See Step 23 and ECHECK Routine in the sample program in Chapter 6, “Sample Program.”

File Operation Functions

The five file operation functions are:

- LSMOS – opens a non-VRAM file on StorHouse. Non-VRAM files are processed sequentially.
- LSMCO creates a new VRAM file on StorHouse and then establishes a data transfer link for writing data to that file.
- LSMOV – opens a StorHouse VRAM file. VRAM files may be processed sequentially, or individual records may be accessed by record number or by key value, depending on the file access method.
- LSMCP – synchronizes file transfer by ensuring that all previously written records have been received and processed by StorHouse.
- LSMCLO – terminates the file operation.

LSMOS, LSMCO, LSMOV, LSMCP, and LSMCLO are described in the following sections.

LSMOS - Open Sequential

LSMOS establishes a data transfer link between the user program and StorHouse, sets the direction of data flow, and identifies StorHouse file being opened. Sequential record transfer operations (read or write) may then be performed on the file.

SETGROUP privilege is required to use LSMOS. For more information about StorHouse privileges, refer to the *Command Language Reference Manual*.

Function Prototype Definition

```
extern int LSMOS  ( struct LSMS_TOKEN *c_token,
                    int message_flag,
                    struct LSMS_TOKEN *o_token,
                    char *mode,
                    char *file_name,
                    long version,
                    struct LSMS_FPW *file_passwords,
                    char *group_name,
                    struct LSMS_FPW *group_passwords,
                    struct LSMS_FLOC *file_location,
                    struct LSMS_ATTR *file_attrib,
                    struct LSMS_OPTS *file_options
                  );
```

Data Structures

```
struct LSMS_FPW
{
   char  read_password[ 9 ];
   char  write_password[ 9 ];
   char  delete_password[ 9 ];
};

struct LSMS_FLOC
{
   char  volumeset_name[ 9 ];
   char  fileset_name[ 9 ];
};
```

```
struct LSMS_ATTR
{
   long  list_size;
   long  file_size;
   long  max_record_len;
   long  transport_flag;
   long  data_xlate_flag;
   long  fixed_record_fl;
   long  cc_ansi_flag;
   long  cc_mach_flag;
   long  block_size;
   long  retention_interval;
};

struct LSMS_OPTS
{
   long  list_size;
   long  lock;
   long  wait;
   long  atf;
   long  edc;
   long  limit;
   long  new; /* This will be newx if compiled with C++. */
   long  unlock;
   long  vtf;
};
```

Argument Description

c_token	A pointer to a session identifier token (connect token).
message_flag	An integer that indicates how indicative text messages are handled. If non-zero (defined value LSMF_MSGHOLD), then text messages from all data transfer errors, including close function errors, are retained in the message stack. If zero (defined value LSMF_NOMSGHOLD), messages are not retrievable after LSMCLO (Close) has been called.
o_token	A pointer to a token (LSMS_TOKEN) structure. The structure need not be initialized. LSMOS fills in this structure with transfer operation identification and clears the asynchronous processing indicator. The application program should not manipulate the members of this structure. It should only be used as the open token (o_token parameter) to other LSMxxx calls for this transfer operation.
mode	A character string that identifies the file reference mode. The acceptable values are READ and WRITE. These values may be specified in upper or lower case.
file_name	A 56-byte character string containing the StorHouse file name or the (operating system dependent) symbolic variable to be referenced. If the symbolic variable is specified, the string must begin with the characters DD=. In this case, the file name

used is the operating system's translation of the string following DD=. If the file_name is longer than 56 characters, LSMOS fails with a return code of 2949.

version
A long integer containing the file's version number. This argument applies only to READ operations. Zero is the default (most current) version. A negative value indicates a relative version number. Positive values are not supported.

file_passwords
A pointer to a structure containing three 9-character (eight data characters plus one byte for null-termination) variables that contain the read, write, and delete passwords, respectively, associated with the file name. The structure member for a password that is not supplied should be set to null. The length of a file/group password (eight characters) is defined value LSML_FILEPW.

group_name
The identifier for the file access group. If the file is stored under the user's default group, this parameter may be omitted; that is, its value may be null. The maximum length for this string is 8 (defined value LSML_GROUPNAME).

group_passwords
A pointer to a password structure containing the read, write, and delete passwords for the group. The structure format is the same as the structure format used for file_passwords.

file_location
A pointer to a file location structure containing the identifiers for the file's destination volume set and file set. This argument applies only to WRITE operations. The file location structure members are:

- volumeset_name – a character string that supplies the name of the file's destination volume set.
- fileset_name – a character string that supplies the name of the file's destination file set.

file_attrib
A pointer to a structure containing a list of long integers that provide file attributes. The caller specifies a value for the first member in the structure, list_size, which contains the number of other members in the structure. The value of block_size should be set to 0. To supply or retrieve all available file attributes, set list_size to 9.

For a mode of WRITE, the caller specifies file attributes. The file_size member is required. All other attributes are optional.

The caller must supply a value for all members included in the structure. Attributes not included in the structure assume a value of 0, which indicates use of the default.

Note the following:

- The value of block_size should be set to 0.
- For a mode of WRITE, the caller specifies file attributes. The file_size member is required. All other attributes are optional.

- For a mode of READ, all file attributes, except block_size and retention_interval, are returned to the caller.
- For flag values, a negative value implies the opposite of the positive value; zero indicates that the default is used.

The file attribute members are:

- list_size – number of other members in the structure.
- file_size – the total file size in bytes. This is an estimate that must be larger than the actual number of bytes being transferred.
- max_record_len – the maximum length for any record in the file.
- transport_flag – a flag value. If positive, the file is in a transportable format that can be retrieved by dissimilar host systems.
- data_xlate_flag – a flag value. If positive, the data is stored as ASCII characters. The data is translated from the native host character set to ASCII when the file is stored. The data is translated from ASCII to the native character set of the receiving host when retrieved. This allows the use of character files on host systems with non-ASCII character sets.
- fixed_record_fl – a flag value. If positive, the records are fixed-length records.
- cc_ansi_flag – a flag value. If positive, the first character of each record is a print carriage control character of the FORTRAN (or ANSI) type.
- cc_mach_flag – a flag value. If positive, the first character of each record is a machine-specific print carriage control character.
- block_size – a user-defined value that specifies the size in bytes of a buffer area used by the Callable Interface. This element is not used and should be set to 0.
- retention_interval – the retention period for the file. Valid values are:
  - Number of days specified as a non-zero, positive integer (for example, 60).
  - LSMV_RETEN_FOREVER, which indicates infinite retention.
  - LSMV_RETEN_ZERO, which indicates the file has no retention period and can be deleted.
  - LSMV_RETEN_DEFAULT, which indicates the retention period is not specified at the file level and assumes the default value.

- If the file's resident file set has a retention attribute equal to FOREVER, ZERO, or a specified number of days, the file set retention attribute determines the file retention attribute.

- If the file's resident file set has a retention attribute of DEFAULT, the RETENTION_MODE system parameter determines the file retention attribute. If RETENTION_MODE is set to BASIC, the file retention is ZERO. If RETENTION_MODE is set to STRICT, the file retention is FOREVER.

file_options A pointer to a structure containing a list of long integers that provide file transfer options. These options correspond to the StorHouse GET and PUT command modifiers. For more information about GET and PUT, refer to the *Command Language Reference Manual*.

Each member is either an integer or a flag value. Integers are positive or zero. Zero indicates that the default value is used. (Note that the actual default value may not equal zero.) Flags are any positive non-zero value (indicates "true" and the option is selected), any negative value (indicates the opposite of "true"), or zero (indicates use of the default).

The caller must supply a value for all attributes included in the structure. Attributes not included in the structure assume a value of 0, which indicates use of the default.

The first member in the structure, list_size, must contain the number of the other members in the structure. To provide all options, set list_size to 8.

The structure members are:

- list_size – number of other members in the structure.
- lock – lock flag. A positive value indicates that the file is to remain explicitly locked after the file operation completes.
- wait – wait for file lock flag.
    - For READ operations, a positive value indicates that the data transfer operation should wait for a locked file to be unlocked.
    - For WRITE operations, this field is no longer used. It is not necessary to change existing code. For new programs, set this field to 0.
- atf – Access Time Factor, a positive integer equal to 1, 2, or 3. This field is used only for WRITE operations.
- edc – Error Detection Code (EDC) identifier, an integer equal to: 1 or 2 to explicitly select the coding algorithm; 0 to indicate use of the default; or negative to generate no codes. A value of zero is recommended. This field is used only for WRITE operations.

- limit – file version LIMIT value, a positive integer between 1 and 32768. This field is used only for WRITE operations.
- new – new file flag. A positive value indicates that a previous version of the file (same group and file name) must not exist in StorHouse. This field is used only for WRITE operations. This will be newx if compiled with C++.
- unlock – unlock flag. This field is obsolete. It is not necessary to change existing code. For new programs, set unlock to 0.
- vtf – Vulnerability Time Factor, an integer equal to 2, 3, or 4. A value of 2 indicates /VTF=NEXT; 3 indicates /VTF=NOW; and 4 indicates /VTF=DIRECT. This field is used only for WRITE operations.

  Refer to the *Command Language Reference Manual* for information about the VTF attribute.

Return Codes

Any Non-Zero Value	The file was not opened. No other Callable Interface functions relating to this file should be issued. In particular, LSMCLO will fail due to an invalid o_token.

Detailed Function Description

LSMOS initiates a file transfer operation between the host and StorHouse. The StorHouse file is identified by the file_name and group_name identifiers. The mode parameter determines the type of processing, either READ or WRITE. The transfer is performed during the session identified by the connect token (c_token).

After the file has been opened successfully (return code zero), read or write functions may be called. The transfer operation is ended by issuing LSMCLO (Close).

LSMOS returns an open token (o_token), which identifies the transfer operation path to subsequent data transfer operations such as read and write. A StorHouse file opened with LSMOS can only be processed sequentially. Facilities implemented by the optional VRAM component, such as reading a record by relative record number, require that the file be created as a VRAM file and that transfer operations be initiated with LSMOV (Open VRAM).

If the message flag is set (non-zero), then LSMMSG must be called after the transfer operation ends (in other words, after LSMCLO [Close] has been called). The dynamic memory allocated for the operation is not released until all messages have been retrieved; that is, until a return code of 3065, indicating no more messages, is received from LSMMSG.

Notes

- If LSMOS returns a non-zero status code, then any associated error messages can be retrieved with the LSMMSG function. The token parameter for LSMMSG should be the connect token, not the open token.

- Refer to Appendix A, “Definition of the LSMxxx Functions,” for programming guidelines on using multiple open statements.

Cross-Reference to Sample Program

See steps 2 and 5 in the sample program in Chapter 6, “Sample Program.”

LSMCO - Create Open

LSMCO creates a new VRAM file on StorHouse and establishes a data transfer link for writing information to that file. LSMCO is equivalent to issuing a StorHouse Command Language CREATE FILE command followed by LSMOV in mode APPEND.

LSMCO requires RECORD privilege. For more information about StorHouse privileges, refer to the *Command Language Reference Manual* in the StorHouse User Document Set.

LSMCO also requires the StorHouse VRAM component.

Function Prototype Definition

```
extern int LSMCO  ( struct LSMS_TOKEN *c_token,
                    int message_flag,
                    struct LSMS_TOKEN *o_token,
                    char *file_name,
                    char *file_password,
                    char *group_name,
                    char *group_password,
                    char *model_file_name,
                    struct LSMS_FLOC *file_location,
                    struct LSMS_CATTR *file_attrib
                  );
```

Data Structures

```
struct LSMS_FLOC
{
   char volumeset_name[ 9 ];
   char fileset_name[ 9 ];
};

struct LSMS_CATTR
{
   long list_size;
   long block_size;
   long checkpt;
   long file_size;
   long data_xlate_flag;
   long atf;
   long cache;
   long edc;
   long limit;
   long vtf;
   long retention_interval;
};
```

Argument Description

c_token	A pointer to a session identifier token (connect token).
message_flag	An integer that indicates how text messages are handled. If non-zero (defined value LSMF_MSGHOLD), then text messages from all data transfer errors, including LSMCLO (close) errors, are retained in the message stack. If zero (defined value LSMF_NOMSGHOLD), messages are not retrievable after LSMCLO has been called.
o_token	A pointer to a token (LSMS_TOKEN) structure. It is not necessary to initialize this structure, because LSMCO sets it to the file identifier. The application program should not manipulate the members of this structure. It should be used only as the o_token parameter to other LSMxxx calls for this transfer operation.
file_name	A 56-byte character string containing the StorHouse file name or the (operating system dependent) symbolic variable to be referenced. If the symbolic variable is specified, the string must begin with the characters DD=. In this case, the file name used is the operating system's translation of the string following DD=. If file_name is longer than 56 characters, LSMCO fails with a return code of 2949.
file_password	An 8-character variable containing the write password for the newly created file. The value of file_password must match the model file's write password (see model_file_name). The read and delete passwords for the new file are copied from the model file's read and delete passwords. If the model file has no write password, file_password should point to a null or all-blanks string. If no model file name is specified, the file_password value becomes the read, write, and delete passwords for the newly created file. The file_password value also supplies the write and delete passwords for any existing version of that file.
group_name	An 8-byte character string that identifies the file access group name for the newly created file and for the model file (see model_file_name). If the file is stored under the account's default group, group_name may point to a null or all-blanks string. SETGROUP privilege is required to specify any group other than the default group.
group_password	An 8-character variable containing the write password for the StorHouse file access group. If no write password is defined for the group, group_password should point to a null or all-blanks string.

model_file_name

A 56-byte character string containing the StorHouse model file name or the (operating system dependent) symbolic variable to be referenced. If the symbolic variable is specified, the string must begin with the characters DD=. In this case, the file name used is the operating system's translation of the string following DD=. If model_file_name is longer than 56 characters, LSMCO fails with a return code of 2949.

The model file must already exist on the StorHouse system. File characteristics for file_name are determined by copying the characteristics of the model file. Non-default values in the file_attrib list override these characteristics.

Only RECORD files can be created without a model file specification. If model_file_name is specified as a blank or null string, then file attributes are determined from the file_attrib list and from system/user defaults.

A prior version of a file cannot be used as a model for a new version of that same file; in other words, model_file_name may not be the same as file_name.

file_location

A pointer to a file location structure containing the identifiers for the file's destination volume set and file set. The file location structure members are:

- volumeset_name – an 8-byte character string that supplies the name of the file's destination volume set.
- fileset_name – an 8-byte character string that supplies the name of the file's destination file set.

If a volume set or file set member is blank or null, the default for the StorHouse account is used. If the account default is also blank, then the specification is copied from the model file.

file_attrib

A pointer to a structure containing a list of long integers that provide file attributes. The caller specifies a value for the first member in the structure, list_size, which contains the number of other members in the structure. To supply or retrieve all available file attributes, set list_size to 10. The value of block_size should be set to 0. The file_size member is required. All other attributes are optional. The caller must supply a value for all members that are included in the list. Attributes not included assume a value of 0.

The values for file_attrib members are either integers or flags. Integer-valued members are either positive or zero. A positive value supplies the specific value used for the parameter. Zero indicates use of the default value.

Note The actual value of the default may not equal 0.

Flags have one of three values:

- Positive indicates true (the option is selected).
- Negative indicates false (the option is not selected).
- Zero indicates use of the default value.

Non-default values override attributes that are determined from the model file. If no model file name is specified, non-default values override normal StorHouse file attribute defaults.

The members of the LSMS_CATTR structure are:

- list_size – the number of other members in the list.

- block_size – a user-defined value that specifies the size in bytes of a buffer area used by the Callable Interface. This member should be set to 0.

- checkpt – a caller-supplied value that indicates the checkpoint number where file processing should be restarted. A value of 0 indicates normal (non-restart) operations. If a non-zero value is specified, the remaining structure members are ignored.

- file_size – the number of bytes of storage space (in units of 1000 bytes) allocated whenever a file is opened for an append operation and whenever a checkpoint is issued. The value must contain enough space for the largest extent set that is written. This extent set includes a data extent, a DF extent, and for KEYED files, a K extent. A file size must always be specified (non-zero) for file creation (in other words, checkpt=0). Refer to Appendix B, “CREATE FILE Command,” for information about specifying file size.

- data_xlate_flag – a flag value. If positive, data is stored as ASCII characters. Data is translated from the native host character set to ASCII when the file is stored and translated from ASCII to the native character set of the receiving host when data is retrieved. This allows use of character files on host systems with non-ASCII character sets.

- atf – the StorHouse Access Time Factor (ATF). Values may equal 1, 2, or 3. Refer to the *Command Language Reference Manual* in the StorHouse User Document Set for more information about ATF.

- cache – the number of sequential records that VRAM caches for an LSMRS, LSMRR, or an LSMRK function for this file when it is opened with an access mode of READ or UPDATE and an access method including RECORD and/or KEYED. The system can use this cache to optimize subsequent sequential reads.

- edc – a flag value. A positive value indicates that error detection coding (EDC) is enabled. A negative value indicates that error detection coding is disabled.

- limit – file version limit value, which may equal a positive number between 1 and 32768. A value of 0 indicates use of the default limit value.

- vtf – Vulnerability Time Factor (VTF), which may equal 2, 3, or 4. A value of 2 indicates VTF=NEXT; 3 indicates VTF=NOW; and 4 indicates VTF=DIRECT. Refer to the *Command Language Reference Manual* in the StorHouse User Document Set for more information about VTF.

- retention_interval – the retention period for the file. Valid values are:

  - Number of days specified as a non-zero, positive integer (for example, 60).

  - LSMV_RETEN_FOREVER, which indicates infinite retention.

  - LSMV_RETEN_ZERO, which indicates the file has no retention period and can be deleted.

  - LSMV_RETEN_DEFAULT, which indicates the retention period is not specified at the file level and assumes the default value.

    - If the file's resident file set has a retention attribute equal to FOREVER, ZERO, or a specified number of days, the file set retention attribute determines the file retention attribute.

  - If the file's resident file set has a retention attribute of DEFAULT, the RETENTION_MODE system parameter determines the file retention attribute. If RETENTION_MODE is set to BASIC, the file retention is ZERO. If RETENTION_MODE is set to STRICT, the file retention is FOREVER.

Return Codes

2629 Specifies that the caller supplied an invalid checkpoint number.

2635 May be caused by the following errors:

- LSMCO was used to create a new version of the model file.
- The specified model file is open for write or update by another user.
- A user tried to create-open a file whose highest version was already in use.

Refer to the message text retrieved by LSMMSG to identify the specific cause of error.

Any Other Non-Zero Value Indicates that the file was not created and is not open. Any other StorHouse functions relating to this file should not be issued. In particular, LSMCLO will fail because of an invalid o_token.

Function Description

LSMCO creates a VRAM file on the StorHouse system and builds an open data transfer path to allow write operations to this new file. The value of file_name identifies the VRAM file. The c_token is the session identifier returned by LSMCON. LSMCO returns a file identifier in the o_token structure. After a successful LSMCO, users may perform operations for the newly created file.

Notes

- Each LSMCO establishes another transfer link and returns another file identifier (o_token). It is the user's responsibility to maintain the integrity of the open tokens.
- If the amount of space indicated by file_size cannot be allocated, StorHouse returns an error code. Refer to Appendix B, "Definition of the LSMxxx Functions" and the *Command Language Reference Manual* (CREATE FILE command description) for information about how to estimate VRAM file sizes.
- A non-zero return code indicates that there may be associated error messages. These messages may be retrieved using LSMMSG. Specify the c_token rather than the o_token in the LSMMSG call.
- Generally model files should be created only for use as models, not for use as data files. When a file is used as a model, it is referenced (mounted) as part of LSMCO processing. If the model is on optical, a physical platter mount may be required. Allocating models as empty files on level F storage prevents this extra platter mount.
- Appendix A, "Definition of the LSMxxx Functions," contains programming guidelines for using multiple open statements and checkpoints. These guidelines also apply to LSMCO.

Cross-Reference to Sample Program

There is no cross-reference to the sample program contained in Chapter 6, "Sample Program."

LSMOV - Open VRAM

LSMOV establishes a data transfer link between the application program and StorHouse, sets the direction of the data flow, indicates the type of processing to be performed, and identifies the StorHouse file being opened. Sequential or record-oriented data transfer operations may then be performed on the file.

LSMOV requires the StorHouse VRAM software component.

Function Prototype Definition

```
extern int LSMOV  ( struct LSMS_TOKEN *c_token,
                    int message_flag,
                    struct LSMS_TOKEN *o_token,
                    char *mode,
                    char *access_method,
                    char *file_name,
                    long revision,
                    struct LSMS_VPW *file_passwords,
                    char *group_name,
                    struct LSMS_VPW *group_passwords,
                    long rel_rec_num,
                    struct LSMS_VATTR *file_attrib
                  );
```

Data Structures

```
struct LSMS_VPW
{
   char  read_password[ 9 ];
   char  write_password[ 9 ];
};

struct LSMS_VATTR
{
   long  list_size;
   long  max_record_len;
   long  last_phy_rec_num;
   long  last_log_rec_num;
   long  file_revision_num;
   long  file_type;
   long  block_size;
   long  version;
   long  checkpt;
};
```

Argument Description

c_token	A pointer to a session identifier token (connect token).
message_flag	An integer that specifies how indicative text messages are handled. If non-zero (defined value LSMF_MSGHOLD), then text messages from all data transfer errors, including close function errors, are retained in the message stack. If zero (defined value LSMF_NOMSGHOLD), messages are not retrievable after close has been called.
o_token	A pointer to a token (LSMS_TOKEN) structure. The structure need not be initialized. It is filled in by LSMOV to contain transfer operation identification and to clear the asynchronous processing indicator. The application program should not manipulate the members of this structure. The structure should only be used as the open token (o_token parameter) to other LSMxxx calls for this transfer operation.
mode	A character string that identifies the file reference mode. The acceptable values are READ, APPEND, and UPDATE. These values may be upper or lower case.
access_method	A character string that identifies the type of file processing. The acceptable values are SEQUENTIAL, KEYED, RECORD, ALL, or a combination of any two or three of SEQUENTIAL, KEYED, and RECORD, separated by commas. These values may be upper or lower case. Specify KEYED or ALL for KRA-type VRAM files only. If you specify KEYED or ALL for RRA-type VRAM files, LSMOV fails. The access_method argument is ignored when mode is set to APPEND.
file_name	A 56-byte character string containing the StorHouse file name or the (operating system dependent) symbolic variable to be referenced. If the symbolic variable is specified, the string must begin with the characters DD=. In this case, the file name used is the operating system's translation of the string following DD=. If the file_name is longer than 56 characters, LSMOV fails with a return code of 2949.
revision	An integer set by the user to indicate the file version's revision number. • Zero is the default (most current) revision. • A positive integer indicates an absolute revision number. • A negative integer indicates a relative revision number. It is the user's responsibility to keep track of absolute revision numbers.
file_passwords	A pointer to a structure containing two 9-character (eight data characters plus one byte for null-termination) variables that contain the read and write passwords, respectively, associated with the file name. The structure member for a password that is not being supplied must be set to null. The length of a file/group password (eight characters) is defined value LSML_FILEPW.
group_name	The identifier for the file access group. If the file is stored under the user's default group, this parameter may be omitted; that is, its value must be all blanks. The maximum length for this string is 8 (defined value LSML_GROUPNAME).

SETGROUP privilege is required to use LSMOV unless the user's default group is used. For more information about StorHouse privileges, refer to the *StorHouse Concepts and Facilities Manual* and the *Command Language Reference Manual* in the StorHouse User Document Set.

group_passwords A pointer to a password structure containing the read and write passwords for the group. The structure format is the same as the structure format used for file_passwords.

rel_rec_num A long integer containing the relative record number of the first record to be read from StorHouse. This value is only meaningful when mode is READ and access_method is SEQUENTIAL.

file_attrib A pointer to a structure containing a list of long integers that provide file attributes. The caller must set the first entry in the structure, list_size, to the number of other members in the structure. The minimum number allowed is 1. The caller also specifies a value of 0 for block_size. The caller may specify a value for file_version and, when applicable, checkpt.

All other file attribute values are returned to the caller when the file is opened. The checkpt value is returned to the caller only if:

- The caller supplies a zero checkpt value, *and*
- The caller attempts to open a checkpointed, software disabled file with LSMOV, mode=APPEND.

In this case, the returned value in checkpt is the file's last checkpoint number.

The file attribute members are:

- list_size – the number of other elements in the structure.
- max_record_len – the maximum length for any record in the file.
- last_phy_rec_num – the last physical record number in the file.
- last_log_rec_num – the last logical record number in the file. If records are deleted from the end of the file, the last logical record number is less than the last physical record number.
- file_revision_num – the absolute revision number of the file version.
- file_type – the VRAM file type. A value of 0 indicates an RRA file, and a value of 1 indicates a KRA file. VRAM file type is specified when the file is created on the StorHouse system with the StorHouse Command Language CREATE FILE command. (For information about the CREATE FILE command, refer to Appendix B, "CREATE FILE Command.")

- block_size – a user-defined value that specifies the size in bytes of a buffer area used by the Callable Interface. This element is not applicable to VMS™ systems and should be set to 0.
- version – a user-supplied value that indicates the version of the file to be opened.
  - To open the latest version, omit the attribute or supply a 0, which is the default.
  - To open a specific version, supply its relative version number as a negative number (-1 through -32767).
  - Positive values are not supported.
- checkpt– a value *supplied* by the caller at open (mode=APPEND) to indicate the checkpoint number where file processing should be restarted. If 0 or omitted, a normal (nonrestart) LSMOV occurs.

  After LSMOV is issued, the value of checkpt is *returned* to the caller only if mode=APPEND, the file being opened is checkpointed and software disabled, *and* the caller set checkpt to 0.

Return Codes

2629	The caller supplied an invalid checkpoint number.
2630	The file was not opened because it is software disabled. A valid checkpoint exists. The last checkpoint number is returned in checkpt.
2636	The caller supplied a checkpoint number but mode was not APPEND.
2637	The caller attempted to open a noncurrent revision of a file at a checkpoint. Only the current revision of a file may be opened at a checkpoint.
Any Other Non-Zero Value	The file was not opened. In this case, no other Callable Interface functions relating to this file should be issued. In particular, do not issue LSMCLO (Close).

Detailed Function Description

LSMOV initiates a file transfer operation between the host and StorHouse for the VRAM file identified by the file name and group identifier. The type of processing is determined by the access_method and mode parameters. The session under which the transfer is to be performed is identified by the connect token. After the file has been opened, other data transfer functions may be called. The transfer operation is ended by calling LSMCLO (Close).

LSMOV returns an open token, which identifies the transfer operation path to subsequent data transfer operations, such as read record or update.

If the message flag is set (non-zero), then the LSMMSG function must be called after the transfer operation is closed. The dynamic memory allocated for the transfer operation is not released until all messages have been retrieved; that is, until a return code of 3065, indicating no more messages, is received from LSMMSG.

Notes

- Each LSMOV call establishes another transfer link and returns another file identifier (o_token). It is the user's responsibility to maintain the integrity of the file identifiers.
- A VRAM file may be opened with a mode of APPEND either to write records into a newly created (empty) file or to add records to a file that already contains data. The two cases can be distinguished by checking the last_phy_rec_num attribute after open; for a new file, it is set to zero.
- Issuing LSMOV with a mode of APPEND attempts to allocate the amount of space that was specified as the value of the /SIZE modifier on the CREATE FILE command for the file that is currently being opened. If this amount of space cannot be allocated (for example, the file's destination file set is filled and cannot extend), StorHouse returns an error code. Refer to Appendix B, "CREATE FILE Command," for more information about how to estimate VRAM file size.
- If the caller attempts to open a checkpointed, software-disabled file and does not supply a checkpoint number, LSMOV returns a 2630 return code and the last checkpoint number. To open the software-disabled file at the returned checkpoint, the caller can issue another LSMOV (mode=APPEND) and supply the previously returned checkpoint number as the current value of the checkpt attribute.
- Only the current (most recent) revision of a file version may be opened at a checkpoint.

The following example illustrates how logical and physical record numbers change. If the last physical record number in the file is record number 8 and that record is deleted, the last physical record number in the file remains 8. The last logical record number becomes record number 7. New records appended to the file begin at record number 9.

If LSMOV returns a non-zero status code, then any associated error messages can be retrieved with LSMMSG. The token parameter for LSMMSG should be the connect token, not the open token.

Refer to Appendix A, "Definition of the LSMxxx Functions," for programming guidelines on using multiple open statements.

Cross-Reference to Sample Program

See Steps 9, 12, and 15 in the sample program in Chapter 6, “Sample Program.”

LSMCP – Checkpoint

LSMCP synchronizes file transfer by ensuring that all previously written records have been received and processed by StorHouse.

LSMCP requires the VRAM StorHouse software component.

Function Prototype Definition

```
extern int LSMCP  ( struct LSMS_TOKEN *o_token,
                    long *return_ckpt_num
                  );
```

Argument Description

o_token	The file identifier returned by LSMOV or LSMCO.
return_ckpt_num	An integer set by StorHouse to the binary number associated with this checkpoint.

Return Codes

Any Non-Zero Value	The file was not successfully checkpointed. No other operation may be performed against a file that returns an error during LSMCP except LSMCLO.

Detailed Function Description

LSMCP synchronizes file transfer to ensure that all records have been written to StorHouse.

LSMCP returns the checkpoint number (value of return_ckpt_num) that must be used to restart the file transfer operation at this position. A data transfer operation (mode of APPEND only) can be restarted by specifying this checkpoint number in the LSMOV (Open VRAM) function (checkpt parameter) or in the LSMCO (Create Open) function (checkpt parameter).

Notes

- To perform LSMCP using LSMOV, the value of mode in the LSMOV call must have been set to APPEND. The access-method is ignored when mode is APPEND.
- Refer to Appendix A, “Checkpoint/Restart and Programming Guidelines,” for information about using LSMCP and LSMOV.

Cross-Reference to Sample Program

There is no cross-reference to the sample program contained in Chapter 6, “Sample Program.”

LSMCLO – Close

LSMCLO terminates a data transfer operation that was initiated by LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM).

Function Prototype Definition

```
extern int LSMCLO  ( struct LSMS_TOKEN *o_token,
                     int xfer_abort_flag
                   );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOS, LSMCO, or LSMOV.
xfer_abort_flag	An integer that indicates whether this is a normal close or a close that is issued to abort the data transfer operation.

- A zero value (defined value LSMF_NORMAL) indicates a normal close.
- A non-zero value (defined value LSMF_ABORT) indicates that StorHouse should abort the data transfer operation. This prevents the file from being cataloged on StorHouse and purges any buffers that may be in transit. The xfer_abort_flag is set (non-zero) when the data stream to StorHouse must be terminated due to an error.

This flag should be set to LSMF_ABORT only for write operations.

Return Codes

For a file write operation, a non-zero return code indicates that the file cannot be guaranteed to be stored in StorHouse.

For a sequential read operation, close is expected to be issued after end-of-file status has been returned to a read or read sequential function. Otherwise, close returns the code 2189; this is *not* an error if the application intended to close before end-of-file.

If xfer_abort_flag is set to LSMF_ABORT, the host abruptly terminates the data transfer link. LSMCLO returns an error code indicating this.

Detailed Function Description

For a sequential write operation, LSMCLO indicates end-of-file. All “in-transit” data buffers are written to StorHouse, and transfer end is signaled. StorHouse completes file storage and directory update operations, and honors the requested VTF level prior to returning operation status. A return code of zero from LSMCLO indicates that the file has been stored in StorHouse.

For a sequential read operation, LSMCLO terminates the transfer and flushes any “in-transit” data buffers. A non-zero return code indicates that all data from the file has not been delivered to the application program.

For record-oriented transfers, LSMCLO causes completion of all file and index updates. A return code of zero indicates that the file state in StorHouse is synchronized with the state expected by the application program.

LSMCLO always releases all StorHouse resources used for the transfer operation. If the message_flag was clear (zero) in the LSMOS, LSMCO, or LSMOV function call, then LSMCLO also releases all host resources used by the transfer. Otherwise, LSMMSG must be called to retrieve all indicative text messages before host resources are completely released.

Notes

- When closing a VRAM file, some record pointers or data must still be posted to the file. If the allocated file size is too small, the file becomes software disabled, and data that was written to the file is lost. Refer to Appendix B, “CREATE FILE Command,” for more information about how to estimate VRAM file size.

- If the message flag was set when the session was established (reference the LSMOS, LSMCO, and LSMOV function descriptions), then the LSMMSG function should be called until it returns a return code of 3065, indicating no more messages.

Cross-Reference to Sample Program

See Steps 4, 7, 11, 14, 22, and ECHECK Routine in the sample program in Chapter 6, “Sample Program.”

Data Transfer Control Functions

Data transfer control functions can be performed once a session has been established and files have been opened. These functions are:

- LSMR – requests the next sequential record of a non-VRAM file from StorHouse.
- LSMRS – requests the next sequential record from a VRAM file.
- LSMRR – requests a record from a VRAM file. The record is identified by its relative record number.
- LSMRK – retrieves a record from a VRAM file. The record is identified by user-supplied key information.
- LSMRNK – requests the next key entry sequenced record from a VRAM file.
- LSMW – sends a record to StorHouse.
- LSMWK – transfers an external key record and a data record to StorHouse.
- LSMDEL – deletes the last record read from a VRAM file.
- LSMCH – changes the last record read in a VRAM file.

The following sections describe these functions.

LSMR – Read

LSMR transfers the next sequential data record from a non-VRAM StorHouse file to the caller's buffer.

Function Prototype Definition

```
extern int LSMR  ( struct LSMS_TOKEN *o_token,
                   char *buffer,
                   long buffer_size,
                   long *return_rec_len
                 );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOS.
buffer	A pointer to an area where the data record is placed. The data record is treated as an array of bytes and is *not* null-terminated.
buffer_size	A long integer value indicating the size, in bytes, of the area pointed to by the buffer parameter.
return_rec_len	A pointer to a long integer that will be set to the length, in bytes, of the record read.

Return Codes

5650	Indicates that end-of-file was encountered. No data record is returned.
2188	Indicates that the buffer supplied for the record is too small. A truncated record was returned. This is a warning only; it is possible to continue the data transfer operation.
Any Other Non-Zero Value	A record was not successfully read.

Detailed Function Description

LSMR reads from StorHouse file identified in the LSMOS (Open Sequential) call that initialized the open token. The data record is placed into the user-supplied buffer, and the length of that record is returned in the area pointed to by the return_rec_len parameter. The mode must be set to READ.

LSMR can only be used for non-VRAM files. For a VRAM file (opened with LSMOV), the LSMRS (Read Sequential) function must be used.

Note

LSMR updates a file's sequential record position. For more information about file positioning, refer to Chapter 3, "Definition of the LSMxxx Functions."

Cross-Reference to Sample Program

See Step 6 in the sample program in Chapter 6, "Sample Program."

LSMRS – Read Sequential

LSMRS requests the next sequential record from a VRAM file.

LSMRS requires the StorHouse VRAM software component.

Function Prototype Definition

```
extern int LSMRS ( struct LSMS_TOKEN *o_token,
                   char *buffer,
                   long buffer_size,
                   long *return_rec_len,
                   long *return_rec_num
                 );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOV.
buffer	A pointer to an area where the data record is placed. The data record is treated as an array of bytes and is *not* null-terminated.
buffer_size	A long integer value indicating the size, in bytes, of the area pointed to by the buffer parameter.
return_rec_len	A pointer to a long integer that will be set to the length, in bytes, of the record read.
return_rec_num	A pointer to a long integer that will be set to the record number of the record read.

Return Codes

5650	End-of-file was encountered. No data record is returned.
2188	The buffer supplied for the record is too small. A truncated record is returned. This is a warning only; it is possible to continue the data transfer operation.
Any Other Non-Zero Value	A record was not successfully read.

Detailed Function Description

LSMRS reads the next sequential record from the VRAM file identified in the LSMOV (Open VRAM) call that initialized the open token. The data record is placed into the user-supplied buffer. The length of that record is returned in the area pointed to by the return_rec_len parameter. The record number is returned in the area pointed to by the return_rec_num parameter.

Notes

- LSMRS updates a file's sequential record position. For more information about file positioning, refer to Chapter 3, "File Positioning."
- To perform LSMRS, the mode for LSMOV must be set to UPDATE or READ. The access_method must include SEQUENTIAL.

Cross-Reference to Sample Program

See Step 13 in the sample program in Chapter 6, "Sample Program."

LSMRR – Read Record

LSMRR requests a specific record from a VRAM file. The record is identified by its relative record number.

LSMRR requires the StorHouse VRAM software component.

Function Prototype Definition

```
extern int LSMRR  ( struct LSMS_TOKEN *o_token,
                    char *buffer,
                    long buffer_size,
                    long *return_rec_len,
                    long *rel_rec_num
                  );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOV.
buffer	A pointer to an area where the data record is placed. The data record is treated as an array of bytes and is *not* null-terminated.
buffer_size	A long integer value indicating the size, in bytes, of the area pointed to by the buffer parameter.
return_rec_len	A pointer to a long integer that will be set to the length, in bytes, of the record read.
rel_rec_num	A long integer value that specifies the record number of the record to be read.

Return Codes

2188	The buffer supplied for the record is too small. A truncated record is returned. This is a warning only; it is possible to continue the data transfer operation.
2587	The record number was out of range. The record could not be found. No data record is returned.
2588	The record with the requested record number was deleted from the file. No data record is returned.
Any Other Non-Zero Value	A record was not successfully read.

Detailed Function Description

LSMRR reads the specific record with the relative record number supplied by the rel_rec_num parameter. The VRAM file must have been opened with LSMOV (Open VRAM). The record that is read is placed into the user-supplied buffer. LSMRR returns the length of the record.

Notes

- LSMRR updates a file's sequential record position. For more information about file positioning, refer to Chapter 3, "File Positioning."
- To perform LSMRR, the mode for LSMOV (Open VRAM) must be set to UPDATE or READ. The access_method must include RECORD.

Cross-Reference to Sample Program

See Step 17 in the sample program in Chapter 6, "Sample Program."

LSMRK – Read Keyed

LSMRK retrieves a record from a VRAM file. The record is identified by user-supplied key information.

LSMRK requires the StorHouse VRAM and KRA software components.

Function Prototype Definition

```
extern int LSMRK  ( struct LSMS_TOKEN *o_token,
                    char *buffer,
                    long buffer_size,
                    long *return_rec_len,
                    char *key_name,
                    char *key_value,
                    long key_length,
                    long *return_rec_num
                  );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOV.
buffer	A pointer to an area where the data record is placed. The data record is treated as an array of bytes and is *not* null-terminated.
buffer_size	A long integer value indicating the size, in bytes, of the area pointed to by the buffer parameter.
return_rec_len	A pointer to a long integer that will be set to the length, in bytes, of the record to be read.
key_name	A character string containing the name of the key used to locate the record. The maximum length for this string is 56 (defined value LSML_KEYNAME). An example of a key_name is LASTNAME.
key_value	A pointer to an area containing the value of the key used for the record search. This area is an array of binary-valued bytes. It is *not* a null-terminated string. An example of a key_value is Kelly.
key_length	A long integer that provides the length of the value in the area pointed to by the key_value parameter. The maximum length is 254 bytes (defined value LSML_KEY).
return_rec_num	A pointer to a long integer that will be set to the record number of the record read.

Return Codes

2188	The buffer supplied for the record is too small. A truncated record is returned. This is a warning only; it is possible to continue the data transfer operation.
2587	The record was not found. There are no records with the supplied key in the file. No data record is returned.
2588	The requested record was deleted from the file. No data record is returned.
Any Other Non-Zero Value	A record was not successfully read.

Detailed Function Description

LSMRK reads a single record from the VRAM file identified in the LSMOV (Open VRAM) call that initialized the open token. The record is identified by the key parameters key_name and key_value. The record that is read is placed into the user-supplied buffer. LSMRK also returns the length of the record.

Notes

- LSMRK updates a file's sequential and key record positions. For more information about file positioning, refer to Chapter 3, "File Positioning."
- To perform this function, the mode for LSMOV (Open VRAM) must be set to UPDATE or READ. The access_method must include KEYED.

Cross-Reference to Sample Program

See Steps 16, 18, 20, and 21 in the sample program in Chapter 6, "Sample Program."

LSMRNK – Read Next Key

LSMRNK requests the next key entry sequenced record from a VRAM file. LSMRNK requires the StorHouse VRAM and KRA software components.

Function Prototype Definition

```
extern int LSMRNK  ( struct LSMS_TOKEN *o_token,
                     char *buffer,
                     long buffer_size,
                     long *return_rec_len,
                     long *return_rec_num
                   );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOV.
buffer	A pointer to an area where the data record is placed. The data record is treated as an array of bytes and is *not* null-terminated.
buffer_size	A long integer value indicating the size, in bytes, of the area pointed to by the buffer parameter.
return_rec_len	A pointer to a long integer that will be set to the length, in bytes, of the record read.
return_rec_num	A pointer to a long integer that will be set to the record number of the record read.

Return Codes

5650	End-of-file was encountered; no data record is returned.
2188	The buffer supplied for the record is too small. A truncated record is returned. This is a warning only; it is possible to continue the data transfer operation.
Any Other Non-Zero Value	A record was not successfully read.

Detailed Function Description

LSMRNK retrieves the next key entry sequenced record from the VRAM file that was previously opened with LSMOV (Open VRAM). The record is placed in the user-supplied buffer. This function returns the length and the record number of the record.

Notes

- LSMRNK updates a file's sequential and key record positions. For more information about file positioning, refer to Chapter 3, "File Positioning."
- To perform LSMRNK, the mode specified to LSMOV (Open VRAM) must be set to UPDATE or READ. The access_method must include KEYED.

Cross-Reference to Sample Program

See Step 18 in the sample program in Chapter 6, "Sample Program."

LSMW – Write

LSMW transfers the next sequential data record from the caller's buffer to a StorHouse file.

Function Prototype Definition

```
extern int LSMW  ( struct LSMS_TOKEN *o_token,
                   char *buffer,
                   long record_length,
                   long *return_rec_num
                 );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOS, LSMCO, or LSMOV.
buffer	A pointer to an area containing the record to be written. The data record is considered as an array of bytes; null terminators are ignored.
record_length	A long integer value that specifies the number of bytes of data to be written.
return_rec_num	A pointer to a long integer that is set to the number of the record that is written. This value is set by StorHouse software.

Return Codes

2210	Is a warning that the record is too short. For a KEYED file, this warning is returned if the record is too short to contain all of its key fields.
Any Other Non-Zero Value	Indicates that the record was not written to StorHouse.

Detailed Function Description

LSMW writes a record to a file in StorHouse that is identified by the LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM) call that initialized the open token. The record is copied from the user-supplied buffer to an internal buffer maintained by the Callable Interface. When the internal buffer fills, it is transmitted to StorHouse.

The record that is written is always assigned the next sequential record number for the file.

Notes

- LSMW moves data from the user record area (buffer) to internal buffers controlled by the Callable Interface. LSMW may return to the caller without actually transferring all user data to StorHouse. Therefore, you can guarantee that the data is stored in StorHouse only after a successful LSMCLO (Close) or LSMCP (Checkpoint).
- Any insufficient space error during a write to a VRAM file leaves the file software disabled. Any data that was written to the file is lost.
- To perform this function, you must either call LSMOV (Open VRAM) with mode set to APPEND or call LSMCO (Create Open). The access_method parameter is ignored.
- For a sequential file opened with LSMOS (Open Sequential), the value of the mode parameter must be set to WRITE.
- LSMW and LSMWK may be used in the same session.

Cross-Reference to Sample Program

See Steps 3 and 10 in the sample program in Chapter 6, “Sample Program.”

LSMWK – Write Key

LSMWK transfers an external key record and a data record to StorHouse. LSMWK requires the StorHouse VRAM and KRA software components.

Function Prototype Definition

```
extern int LSMWK  ( struct LSMS_TOKEN *o_token,
                    char *buffer,
                    long record_length,
                    char *key,
                    long key_length,
                    long *return_rec_num
                  );
```

Argument Description

o_token	A pointer to an open token initialized by LSMCO or LSMOV.
buffer	A pointer to an area containing the data record to be written to StorHouse. The data record is considered an array of bytes; null terminators are ignored.
record_length	A long integer value that specifies the number of bytes of data to be written.
key	A pointer to an area containing the external key. This area is not a null-terminated string; it is an array of binary-valued bytes.
key_length	A long integer that provides the length, in bytes, of the external key in the area pointed to by key.
return_rec_num	A pointer to a long integer that is set by StorHouse to the record number of the last data record written to StorHouse.

Return Codes

2210	A warning indicating that the external key record is too short to contain all of its key fields.
Any Other Non-Zero Value	The record was not written to StorHouse.

Detailed Function Description

LSMWK writes an external key record and a data record to StorHouse file that is identified by the o_token returned by LSMOV or LSMCO. The file must have been created either with the CREATE FILE command using the /EXTERNAL modifier, or with LSMCO using a model file with external keys. Refer to Appendix B, “CREATE FILE Command,” for information about the CREATE FILE command.

The record is copied from the user-supplied buffer to an internal buffer maintained by the Callable Interface.

The data record becomes the next sequential record in the file.

Notes

- LSMWK moves data from the user record area (buffer) to internal buffers controlled by the Callable Interface. LSMWK may return to the caller without actually transferring all user data to StorHouse. Therefore, data can only be guaranteed to be stored in StorHouse after a successful LSMCLO or LSMCP.

- To perform LSMWK, the mode parameter in LSMOV must be set to APPEND. The access_method parameter is ignored.

- The Callable Interface considers an LSMWK with a specified key_length of 0 the same as an LSMW. LSMW and LSMWK may be used in the same session.

- To write a data record and no external key record, specify a value of 0 for the key_length field when you issue LSMWK. To write one external key record and five associated data records, issue LSMWK to write the external key record and the first data record. Then issue four LSMWKs with a specified key_length of 0 to write the remaining four data records.

- The actual external key record cannot be accessed by an application. Key information is extracted from the record and stored in a key data base on StorHouse. Therefore, users cannot change an external key record once it is written or read the external key file to determine its keys.

- By definition, external keys are external to, or not part of, the data record. Therefore, data records associated with a given external key should contain control information that allows an application to determine when it has processed the last data record belonging to that external key.

- Any insufficient space error during a write to a VRAM file leaves the file software disabled. Any data that was written to the file since the last close or checkpoint is lost.

Cross-Reference to Sample Program

There is no cross-reference to the sample program contained in Chapter 6, “Sample Program.”

LSMDEL – Delete Record

LSMDEL deletes the last record read from a VRAM file. LSMDEL requires the StorHouse VRAM software component.

Function Prototype Definition

```
extern int LSMDEL ( struct LSMS_TOKEN *o_token);
```

Argument Description

o_token	A pointer to an open token initialized by LSMOV.

Return Codes

2612	An attempt was made to delete a record without reading the record first.
Any Other Non-Zero Value	The record was not deleted.

Detailed Function Description

LSMDEL logically removes the last record read from the VRAM file that was previously opened with LSMOV (Open VRAM). The file is identified by the open token initialized by the LSMOV call.

Note

To perform this function, the mode for LSMOV (Open VRAM) must be set to UPDATE. For a more complete description of mode and the associated access_method parameter, refer to the LSMOV function description on page 5-20.

Cross-Reference to Sample Program

See Step 20 in the sample program in Chapter 6, “Sample Program.”

LSMCH – Change Record

LSMCH logically changes the last record read from a VRAM file by transmitting a replacement record to StorHouse.

LSMCH requires StorHouse VRAM software component.

Function Prototype Definition

```
extern int LSMCH  ( struct LSMS_TOKEN *o_token,
                    char *buffer,
                    long record_length
                  );
```

Argument Description

o_token	A pointer to an open token initialized by LSMOV.
buffer	A pointer to an area containing the record to be written. The data record is considered as an array of bytes; null terminators are ignored.
record_length	A long integer value that specifies the number of bytes of data to be written.

Return Codes

2612	An attempt was made to change a record without reading the record first.
Any Other Non-Zero Value	The record was not changed.

Detailed Function Description

LSMCH logically replaces the last record read from the VRAM file that was previously opened with LSMOV (Open VRAM). The file is identified by the open token initialized by the LSMOV call.

Note

To perform this function, the mode for LSMOV (Open VRAM) must be set to UPDATE. For a more complete description of mode and the associated access_method parameter, refer to the LSMOV function description.

Cross-Reference to Sample Program

See Step 19 in the sample program in Chapter 6, “Sample Program.”

StorHouse Command Submission

There is one StorHouse command submission function: LSMSCI. LSMSCI allows an application to:

- Send selected StorHouse Command Language commands to StorHouse and to retrieve response text from those commands.
- Direct administrative operations from an application rather than from a user at a terminal through the Interactive Interface.

LSMSCI is described in the following section.

LSMSCI – StorHouse Command Interface

LSMSCI sends a text string to StorHouse to be processed as a StorHouse Command Language command. For descriptions of the available commands, refer to the *Command Language Reference Manual*.

Function Prototype Definition

```
extern int LSMSCI  ( struct LSMS_TOKEN *c_token,
                     char *CR_buf,
                     char *resp_buf,
                     long resp_bufsize,
                     struct LSMS_SCIR *resp_info
                   );
```

Data Structures

```
struct LSMS_SCIR
{

   long list_size;
   long length;
   long status;
   long severity;
   long cmd_ended;
   long prompt;
   long suppress;
};
```

Argument Description

c_token A pointer to a session identifier token (connect token).

cr_buf A pointer to a null-terminated character string that is sent to StorHouse. The maximum length of this character string is 255.

This string may be either a StorHouse command or a reply to a previous prompt from StorHouse:

- The first LSMSCI call or any LSMSCI call following a call that returned a “true” cmd_ended flag sends this character string to StorHouse as a command.

- An LSMSCI call following a call that returned a “true” prompt flag sends this character string to StorHouse as a reply to the prompt.

- In any other case, this string is not used, and a null pointer may be specified.

resp_buf
: A pointer to an area where the response text from StorHouse is placed. The response is a character string and is null-terminated.

resp_bufsize
: A long integer value indicating the size, in bytes, of the area pointed to by resp_buf. This value should be no smaller than 132 (defined value LSML_MAXTEXT). If the response buffer is too small to contain the response text, the text is truncated to fit in the supplied buffer, and the return status indicates an error.

The resp_bufsize may not equal zero. A zero value causes a 3022 return code, indicating that a zero buffer size was passed to a StorHouse read function.

resp_info
: A pointer to a structure containing a list of long integers that LSMSCI uses to return information about the StorHouse response. The members in the structure are:

- list_size – the number of other members of the resp_info structure. The caller must set this entry to 6.
- length – the length of the response text, not including the null terminator.
- status – the status code associated with execution of the command. This code is returned only when *command end* is indicated. Refer to the cmd_ended member description below. Note that this code refers to the StorHouse status, while the value returned by LSMSCI indicates the status of the function request processing.
- severity – the severity of the error indicated by the status code (above). This value is between 0 and 20 (see “Return Codes” below).
- cmd_ended – a flag indicating (if 1) that the command has completed execution (ended).
- prompt – a flag indicating (if 1) that the response text is actually a prompt from StorHouse.
- suppress – a flag indicating (if 1) that StorHouse suggests suppression of print or display of any information supplied in response to the prompt. This flag is valid only if the prompt flag is set. This flag usually indicates that StorHouse is prompting for security information, such as a password.

Return Codes

Any Non-Zero Value
: The command text was not processed by StorHouse. The resp_info structure members should not be used.

Zero
: The command was successfully passed to StorHouse, and a response was received. The status associated with the StorHouse command execution is indicated by the status member of the resp_info structure. Severity can be used to examine the general

condition associated with command execution without testing for specific status codes.

The severity codes are:

- 0 – Normal, no errors detected.
- 4 – Warning, results may not be as expected.
- 8 – Error, results are probably incorrect, and corrective action may be required.
- 12 – Severe errors occurred, corrective action is required.
- 16 – Request could not be processed.
- 20 – Hardware or system software error prevented command processing. (Partial execution may have occurred, or StorHouse may have processed the command, but responses may have been lost.)

Detailed Function Description

LSMSCI allows direct user access to the StorHouse command processing facilities by sending user-provided text strings to StorHouse. The first parameter to LSMSCI must point to a connect token for an established session (refer to the function description for LSMCON).

The caller supplies a pointer to a buffer containing the command text, a pointer to a buffer where a response message can be copied, the length of the response buffer, and a pointer to a structure where response status information can be returned. The caller must also set the number of members in the response status structure.

Text messages generated by StorHouse in response to the submitted command are returned to the user-supplied response buffer. A cmd_ended flag indicates when all response messages have been returned. LSMSCI must be called repeatedly until cmd_ended is set.

LSMSCI is only intended for accessing informational commands such as SHOW FILE. GET and PUT commands do not function correctly if issued using this function. Data transfers must be accomplished through the open and read/write functions. In addition, the SET USER command cannot be used to change session defaults. For more information about StorHouse Command Language commands that can be accessed with LSMSCI, consult your StorHouse system administrator or your SGI customer support representative.

Some commands or command options cause StorHouse to request additional information from the host by prompting for a text string. A reply to the prompt is required to complete execution of any such command. The LSMSCI function allows the user to provide a response string to a StorHouse prompt. Whenever StorHouse prompts for additional information, a prompt flag is set in the status returned by LSMSCI, and the returned text is the StorHouse prompt string. The user must place the reply to this prompt in the command text buffer and call LSMSCI to send the contents of that buffer to StorHouse in reply to the prompt.

Notes

- LSMSCI must be called repeatedly until the cmd_ended flag is set. Failure to do so makes the session link unusable.
- Whenever the prompt flag is set, the next call to LSMSCI supplies a response. If the string pointed to by cr_buf is empty (contains only a null terminator), then a null response is sent to StorHouse. In this case, the cr_buf argument may *not* contain a null pointer.

Cross-Reference to Sample Program

See Step 8 in the sample program in Chapter 6, “Sample Program.”

General Usage Functions

The general usage functions are:

- LSMCK – waits for and tests the completion status of any asynchronous operation.
- LSMASY– sets the supplied token so that subsequent function requests are processed in asynchronous mode.
- LSMMSG– retrieves indicative messages associated with a previous LSMxxx function.
- LSMAB – attempts to terminate the last asynchronous request for the supplied token or attempts to terminate a StorHouse command passed to StorHouse with LSMSCI.

These functions are described in the following sections.

LSMCK – Check

LSMCK waits for and tests the completion status of any asynchronous operation.

Function Prototype Definition

```
extern int LSMWK ( struct LSMS_ TOKEN *any_token);
```

Argument Description

any_token A pointer to a token structure. The token structure may be either a connect token (session identifier) initialized by LSMCON (Connect) or an open token (transfer operation identifier) initialized by LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM).

Return Codes

LSMCK returns the final status from the previously issued asynchronous LSMxxx function call.

2974 The prior LSMxxx function was not issued in asynchronous mode.

Detailed Function Description

LSMCK ensures that the previous asynchronous function has completed and returns the completion status from that function. If a c_token is supplied, the check applies to the last function issued on a session link. If an o_token is supplied, the check applies to the last data transfer function associated with that o_token.

Notes

- As explained in the beginning of this chapter, this implementation of the Callable Interface does not provide truly asynchronous capability.
- A function call is asynchronous only if the LSMASY function has been previously called pointing to the same token.

Cross-Reference to Sample Program

There is no cross-reference to the sample program contained in Chapter 6, “Sample Program.”

LSMASY – Set Asynchronous Mode

LSMASY sets the supplied token so that subsequent function requests are processed in asynchronous mode.

Function Prototype Definition

```
extern int LSMASY   ( struct LSMS_TOKEN *any_token,
                      long p1_value,
                      int set_flag
                    );
```

Argument Description

any_token	A pointer to a token structure. The token structure is either a connect token (session identifier) returned by LSMCON (Connect) or an open token (transfer operation identifier) initialized by LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM).
p1_value	A long integer value that provides system-dependent information required for synchronization of operation completion. The use of this value is system dependent and is not applicable to VMS systems.
set_flag	An integer that indicates whether the token is set for asynchronous or synchronous operation. If zero (defined value LSMF_ASYNC), the token is set for asynchronous operation. If non-zero (defined value LSMF_SYNC), the token is reset to the default (synchronous) state.

Return Codes

Zero	The return code from this function is always zero.

Detailed Function Description

LSMASY allows the synchronous/asynchronous state of a token to be changed. LSMCON (Connect), LSMOS (Open Sequential), LSMCO (Create Open), and LSMOV (Open VRAM) initialize a token to the synchronous state, so that all LSMxxx functions issued against that token are synchronous (that is, they return only after the requested operation has been performed). If the application program must issue asynchronous requests, the token state must be set to asynchronous by calling

LSMASY with a set_flag value of LSMF_ASYNC. The token can be restored to the default state by calling LSMASY with set_flag set to LSMF_SYNC.

Application programs that use asynchronous operations are generally portable only if the synchronization code is isolated in modules that are replaced when the program is moved to another computer/operating system.

LSMASY and LSMCK may be used in this implementation, but keep in mind that any LSMxxx function called with the token set to asynchronous mode still returns control only after the requested operation has completed. Refer to the beginning of this chapter for more information.

Cross-Reference to Sample Program

There is no cross-reference to the sample program contained in Chapter 6, “Sample Program.”

LSMMSG – Message Retrieve

LSMMSG retrieves indicative messages associated with a previous LSMxxx function.

Function Prototype Definition

```
extern int LSMMSG  ( struct LSMS_TOKEN *any_token,
                     char *message_buffer,
                     long message_ buffer_size,
                     long *returned_message_len
                   );
```

Argument Description

any_token	A pointer to a token structure. The token structure is either a connect token (session identifier) returned by LSMCON (Connect) or an open token (transfer operation identifier) initialized by LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM).
message_buffer	A pointer to the area where the error message text will be moved. The message is returned as a null-terminated string.
message_buffer_ size	A long integer value that provides the size of the message_buffer area.
returned_message_ len	A pointer to a long-integer that is set to the length of the error message retrieved.

Return Codes

3065	No more messages are available.

Detailed Function Description

LSMMSG retrieves indicative text messages created during execution of the prior LSMxxx function.

- If a c_token is supplied, LSMMSG returns messages from the last session-related function.

- If an o_token is supplied, LSMMSG returns error messages from the last file-related function call.

LSMMSG returns one message in the user-supplied buffer. This function also returns the length of the message.

Notes

- The maximum buffer length required to retrieve an error message is 132 bytes (defined value LSML_MAXTEXT). If the user-supplied buffer is shorter than 132 bytes, messages longer than the length of the supplied buffer are truncated when returned. Messages are null-terminated after the last non-blank character.
- If message_flag was set for LSMCON (Connect), LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM), then LSMMSG must be called after an LSMDIS (Disconnect) or LSMCLO (Close) call or after a failing LSMCON. LSMMSG must be called repeatedly until a return code of 3065, indicating no more messages, is received. Otherwise, dynamically allocated memory used by the Callable Interface functions may not be released.
- The correct token for an LSMMSG call following any type of open is the c_token.

Cross-Reference to Sample Program

See Steps 4, 7, 11, 14, 22, 23, and EMSGS Routine in the sample program in Chapter 6, “Sample Program.”

LSMAB – Abort

LSMAB attempts to terminate the last asynchronous function request for the supplied token or attempts to terminate a StorHouse command passed to StorHouse with LSMSCI.

Function Prototype Definition

```
extern int LSMAB ( struct LSMS_TOKEN *any_token);
```

Argument Description

any_token	A pointer to a token structure. The token structure is either a connect token (session identifier) returned by LSMCON (Connect) or an open token (transfer operation identifier) initialized by LSMOS (Open Sequential), LSMCO (Create Open), or LSMOV (Open VRAM).

Return Codes

2989	No asynchronous function was outstanding and no LSMSCI sequence was active.
2990	The function to be ABORTed was LSMCON (Connect). This is not allowed.

Otherwise, the return code from LSMAB is always zero. LSMCK (Check) must be issued to retrieve the return code associated with ABORT.

Detailed Function Description

LSMAB unconditionally attempts to terminate the last function issued for a session or a data transfer function.

- If a c_token is supplied, the ABORT applies to the last function started on a session link.
- If an o_token is supplied, then the ABORT applies to the last data transfer function associated with that open token.

Notes

- LSMAB can be issued for asynchronous functions or during a sequence of LSMSCI calls, prior to receiving the *command end* indication. For synchronous functions, the operation is complete when control returns to the user program; hence, LSMAB has no effect.

- LSMAB can only request termination of function processing. The function may have already completed or may complete before the ABORT request is forwarded. The return code associated with the original function must be interrogated to determine the actual outcome of the ABORT attempt.

- ABORT is intended as a mechanism to terminate a pending asynchronous operation for a data transfer or session, when that transfer or session is to be subsequently terminated. For some operations, such as sequential read or write functions, an ABORT causes the entire data transfer to fail.

- When LSMAB is issued during an LSMSCI sequence, termination of processing of the command by StorHouse is requested. However, the user must continue calling LSMSCI until *command end* is indicated.

Cross-Reference to Sample Program

There is no cross-reference to the sample program contained in Chapter 6, “Sample Program.”

Chapter
6

Sample Program

The following sample program in C illustrates the use of the LSMxxx subroutine to invoke the StorHouse host Callable Interface. This program:

- Establishes a StorHouse session.
- Creates, opens, and closes sequential and VRAM files (with both internal and external keys).
- Accesses all files in all available modes (READ, WRITE, UPDATE, APPEND) and performs all file operations at least once.
- Performs synchronous and asynchronous functions.
- Checks for errors and function completion.
- Retrieves informational and error messages.
- Disconnects from StorHouse.

If an error occurs, this program closes files and stops. An error message is displayed.

LSMxxx Sample Program

```
/***************************************************************************
*  This program illustrates using the LSMxxx subroutine to invoke the      *
*  StorHouse host Callable Interface.                                      *
*                                                                          *
*                                                                          *
*  If an error occurs, this program closes files and stops. An error       *
*  message is displayed.                                                   *
*                                                                          *
*                                                                          *
*  The program performs the following functions:                           *
*                                                                          *
*                                                                          *
*  Step 1 - connects to StorHouse (LSMCON).                                *
*                                                                          *
*  Step 2 - opens a sequential (non-VRAM) file on StorHouse for writing    *
*  (LSMOS).                                                                *
*                                                                          *
*  Step 3 - writes 100 80-byte records to the sequential file (LSMW).      *
*                                                                          *
*  Step 4 - closes the sequential file (LSMCLO) and retrieves any messages *
*  (LSMMSG).                                                               *
*                                                                          *
*  Step 5 - opens the sequential (non-VRAM) file for reading (LSMRS).      *
*                                                                          *
*  Step 6 - reads 100 records from the sequential file (LSMRS).            *
*                                                                          *
*  Step 7 - closes the sequential file (LSMCLO) and retrieves any          *
*  messages (LSMMSG).                                                      *
*                                                                          *
*  Step 8 - creates a KEYED VRAM file on StorHouse (LSMSCI).               *
*                                                                          *
*  Step 9 - opens the KEYED VRAM file for writing (LSMOV).                 *
*                                                                          *
*  Step 10 - writes 100 80-byte records to the KEYED VRAM file (LSMW)      *
*                                                                          *
*  Step 11 - closes the KEYED VRAM file (LSMCLO) and retrieves any messages*
*  (LSMMSG).                                                               *
*                                                                          *
*  Step 12 - opens the KEYED VRAM file with a mode of READ and a method of *
*  SEQUENTIAL (LSMOV).                                                     *
*                                                                          *
*  Step 13 - reads 100 records in the VRAM file (LSMRS).                   *
*                                                                          *
*  Step 14 - closes the VRAM file (LSMCLO) and retrieves any               *
*  messages (LSMMSG).                                                      *
*                                                                          *
*  Step 15 - opens the VRAM file with a mode of UPDATE and a method        *
*  of ALL (LSMOV).                                                         *
*                                                                          *
*  Step 16 - reads the VRAM file in reverse key order (LSMRK).             *
*                                                                          *
*  Step 17 - reads the VRAM file in reverse record number order (LSMRK).   *
*                                                                          *
*  Step 18 - reads a specific key in the file (LSMRK); then reads the next *
*  key (LSMRNK).                                                           *
*                                                                          *
*  Step 19 - reads a specific key in the file (LSMRK); then changes the    *
*  key (LSMCH).                                                            *
*                                                                          *
*  Step 20 - reads a specific key in the file (LSMRK); then deletes the    *
*  record with the specified key (LSMDEL).                                 *
*                                                                          *
*  Step 21 - reads a specific keyed record and checks that the record has  *
*  been deleted.                                                           *
*                                                                          *
*  Step 22 - closes the KEYED VRAM file (LSMCLO) and retrieves any         *
*  messages (LSMMSG).                                                      *
*                                                                          *
*  Step 23 - disconnects from StorHouse (LSMDIS) and retrieves any error   *
*  messages (LSMMSG).                                                      *
*                                                                          *
***************************************************************************/

#include <stdio.h>
#include "lsmdefs.h"

/***************************************************************************
*                                                                          *
*                           GLOBAL DECLARATIONS                            *
*                                                                          *
***************************************************************************/

/*
 * Tokens returned by LSMCON, LSMOS, and LSMOV.
 */

static struct LSMS_TOKEN ctoken;       /* Connect token for session    */
static struct LSMS_TOKEN otoken;       /* Open token for data transfer */

/*
 * File-is-opened switch
 */

static long file_open;                 /* File-is-opened switch        */

/*
 * Structures used by LSMOS, LSMSCI and LSMOV.
 */

static struct LSMS_FPW filepassword;   /* File passwords               */
static struct LSMS_FPW grouppassword;  /* Group passwords              */
static struct LSMS_FLOC location;      /* Volume set and file set      */
static struct LSMS_ATTR attr;          /* File attributes (LSMOS)      */
static struct LSMS_OPTS options;       /* File transfer options        */
static struct LSMS_SCIR info;          /* StorHouse cmd interface response */
static struct LSM_VATTR vattr;         /* File attributes (LSMOV)      */

/*
 * ROUTINES
 */

static void echeck();                  /* Error checker                */
static void emsgs();                   /* Message retriever            */

/*
 * MAIN PROGRAM
 */

main()
{
   long  i, j;                         /* Scratch registers            */
   long msgflag;                       /* Message flag                 */
   long return_rec_num;                /* Record number written        */
   long return_rec_len;                /* Length of record read        */
   long abort_flag;                    /* Flag for abort option        */
   long return_code;                   /* Completion status            */

   char cmdbuf[60];                    /* Command buffer for LSMSCI    */
   char respbuf[133];                  /* Response buffer for LSMSCI   */

   char buffer[80];                    /* Data buffer area             */
   char cmprbuf[80];                   /* Data buffer area             */

   char group[ LSML_GROUPNAME + 1 ];   /* Group name                   */
   char filename[ LSML_FNAME + 1 ];    /* File  name                   */
   char account[ LSML_AIC + 1 ];       /* Account name                 */
   char password[ LSML_SOPW + 1 ];     /* Account, or signon, password */

/*
 * Initialize some variables.
 */

file_open = 0;                         /* Set to file NOT opened       */
abort_flag = LSMF_NORMAL;              /* Perform a normal close       */
msgflag = LSMF_MSGHOLD;                /* All session msgs returned    */
```

```
/***************************************************************************
*                                                                         *
*                               Step 1                                    *
*                                                                         *
*                 Connect to StorHouse (LSMCON).                          *
*                                                                         *
***************************************************************************/

strcpy( account, "SYSTEM" );           /* StorHouse account to use     */

strcpy( password, "SYSTEM" );          /* StorHouse password to use   */

return_code = LSMCON( &ctoken, msgflag, account, password, "", "" );

emsgs( &ctoken );                      /* Retrieve messages if any     */

echeck( return_code, "LSMCON" );       /* Check for errors             */

/*
 * Group and file passwords are NUL, indicating no passwords.
 * VSET, FSET, and GROUP NAME are NUL, indicating use of the
 * signon account defaults.
 *
 */

strcpy( filepassword.read_password, "" );    /* Password=none          */
strcpy( filepassword.write_password, "" );   /* Password=none          */
strcpy( filepassword.delete_password, "" );  /* Password=none          */

strcpy( groupassword.read_password, "" );    /* Password=none          */
strcpy( groupassword.write_password, "" );   /* Password=none          */
strcpy( groupassword.delete_password, "" );  /* Password=none          */

strcpy( location.volumeset_name, "" );       /* Use account default VSET */
strcpy( location.fileset_name, "" );         /* Use account default FSET */
strcpy( group, "" );                         /* Use account default GROUP*/

/***************************************************************************
*                                                                         *
*                               Step 2                                    *
*                                                                         *
*           Open a sequential (non-VRAM) file for writing (LSMOS).        *
*                                                                         *
***************************************************************************/

strcpy( filename, "sequential_file" );       /* Setup file name        */

attr.list_size = 7;                    /* Number of desired attributes */
attr.file_size = 50000;                /* Maximum file size for WRITE  */
attr.max_record_len = 80;              /* Maximum length for any record*/
attr.transport_flag = 1;               /* File in transportable format */
attr.data_xlate_flag = 1;              /* Data stored in ASCII         */
attr.fixed_record_fl = 1;              /* Records are fixed length     */
attr.cc_ansi_flag = -1;                /* No carriage control character*/
attr.cc_mach_flag = -1;                /* No carriage control character*/

options.list_size = 8;                 /* Number of desired options    */
options.lock = 0;                      /* Do not lock file at close    */
options.wait = 0;                      /* Do not wait for locked file  */
options.atf = 0;                       /* Use system default           */
options.edc = 0;                       /* Use system default           */
options.limit = 0;                     /* Use system default           */
options.new = -1;                      /* A previous version may exist */
options.unlock = 0;                    /* Do not release explicit lock */
options.vtf = 2;                       /* VTF = NEXT                   */

return_code = LSMOS( &ctoken, msgflag, &otoken, "WRITE",
   filename, 0, &filepassword, group, &grouppassword, &location,
   &attr, &options );

emsgs( &ctoken );                      /* Retrieve messages if any     */

echeck( return_code, "LSMOS" );        /* Check for error              */

file_open = 1;                         /* Set to file is opened        */

/****************************************************************************
*                                                                          *
*                                 Step 3                                   *
*                                                                          *
*            Write 100 80-byte records to the file (LSMW).                 *
*                                                                          *
****************************************************************************/

for( i = 0; i < 100; i++ )
{
   sprintf( buffer, "This is record number %3.3d  ", i );

   for( j = 28; j < 80; j++ )
      buffer[j] = ' ';                 /* blank fill record            */

   return_code = LSMW( &otoken, buffer, sizeof(buffer), &return_rec_num);

   echeck( return_code, "LSMW" );
}

/****************************************************************************
*                                                                          *
*                                 Step 4                                   *
*                                                                          *
*                  Close the sequential file (LSMCLO).                     *
*                  Retrieve any messages (LSMMSG).                         *
*                                                                          *
****************************************************************************/
```

```
return_code = LSMCLO( &otoken, abort_flag );

emsgs( &otoken );                      /* Retrieve messages if any     */

echeck( return_code, "LSMCLO" );       /* Check for error              */

file_open = 0;                         /* Set to file NOT opened       */

/**************************************************************************
*                                                                         *
*                               Step 5                                    *
*                                                                         *
*           Open a sequential file for reading (LSMOS).                   *
*                                                                         *
**************************************************************************/

attr.list_size = 0;                    /* Number of desired attributes */

options.list_size = 2;                 /* Number of desired options    */
options.lock = 0;                      /* Do not lock file at close    */
options.wait = 1;                      /* Wait for locked file         */

return_code = LSMOS( &ctoken, msgflag, &otoken, "READ",
   filename, 0, &filepassword, group, &grouppassword, &location,
   &attr, &options );

emsgs( &ctoken );                      /* Retrieve messages if any     */

echeck( return_code, "LSMOS" );        /* Check for error              */

file_open = 1;                         /* Set to file is opened        */

/**************************************************************************
*                                                                         *
*                               Step 6                                    *
*                                                                         *
*           Read the sequential file until return code 5650 (EOF)        *
*           is received (LSMR).  The program should be able to           *
*           read 100 records.                                             *
*                                                                         *
**************************************************************************/

for( i = 0; i < 100; i++ )
{
   sprintf( cmprbuf, "This is record number %3.3d  ", i );

   for( j = 28; j < 80; j++ )
      cmprbuf[j] = ' ';                /* blank fill record            */

   return_code = LSMR( &otoken, buffer, sizeof(buffer), &return_rec_len);

   echeck( return_code, "LSMR" );
   if( strncmp( buffer, cmprbuf, sizeof(buffer) ) != 0 )
      printf( "record data mismatch for record %d.\n", i );
}

return_code = LSMR( &otoken, buffer, sizeof(buffer), &return_rec_len );

if( return_code != 5650 )
   printf( "Expected EOF did not occur.\n" );

/****************************************************************************
*                                                                           *
*                                Step 7                                     *
*                                                                           *
*                  Close the sequential file (LSMCLO).                      *
*                  Retrieve any messages (LSMMSG).                          *
*                                                                           *
****************************************************************************/

return_code = LSMCLO( &otoken, abort_flag );

emsgs( &otoken );                       /* Retrieve messages if any     */

echeck( return_code, "LSMCLO" );        /* Check for error              */

file_open = 0;                          /* Set to file NOT opened       */

/****************************************************************************
*                                                                           *
*                                Step 8                                     *
*                                                                           *
*                    Create a VRAM file using LSMSCI.                       *
*                                                                           *
****************************************************************************/

info.list_size = 6;                     /* Number of desired options    */

strcpy( respbuf, "" );

strcpy( cmdbuf,

   "CREATE FILE VRAMKEYED /SIZE=50K /TYPE=KEYED /REPLACE" );

printf( "%s\n", cmdbuf );

return_code = LSMSCI(&ctoken, cmdbuf, respbuf, sizeof(respbuf), &info);

/*
 * Print out all messages from StorHouse.
 */

while ( return_code == 0 )
{
   printf("%s\n",respbuf);
   strcpy ( respbuf, "" );

   if (info.cmd_ended != 0)
   {

      return_code = info.status;
      break;
   }

   /*
    * Respond to the StorHouse prompt with key definitions
    * for the VRAM file.
    */

   if (info.prompt == 1)
   {
      strcpy( cmdbuf, "KEY KEY1 1:4" );

      printf( "%s\n", cmdbuf );

      return_code = LSMSCI( &ctoken,cmdbuf,respbuf,sizeof(respbuf),&info);

      break;
   }
   else
      return_code = LSMSCI( &ctoken, "", respbuf,sizeof(respbuf),&info);

}

/*
 *  Print out all messages from StorHouse.
 */

while ( return_code == 0 )
{
   printf("%s\n",respbuf);

   strcpy ( respbuf, "" );

   if (info.cmd_ended != 0)
   {
      return_code = info.status;
      break;
   }

   /*
    *  Respond to the StorHouse prompt with the EXIT command to exit
    *  key definition mode.
    */

   if (info.prompt == 1)
   {
      strcpy( cmdbuf, "EXIT" );
      printf( "%s\n", cmdbuf );

      return_code = LSMSCI( &ctoken, cmdbuf, respbuf, sizeof(respbuf), &info);

      break;

   }
   else

      return_code = LSMSCI( &ctoken, "", respbuf, sizeof(respbuf), &info );
}

/*
 *  Print out all messages from StorHouse.
 */

while ( return_code == 0 )
{
   printf("%s\n",respbuf);

   strcpy ( respbuf, "" );

/*
 * Look for end of command, then break out.
 */

   if (info.cmd_ended != 0)
   {
      return_code = info.status;
      break;

   }
   else

      return_code = LSMSCI( &ctoken, "", respbuf, sizeof(respbuf), &info );

}

echeck( return_code, "LSMSCI" );

/****************************************************************************
*                                                                          *
*                                 Step 9                                   *
*                                                                          *
*                                                                          *
*                  Open a VRAM file for writing (LSMOV).                   *
*                                                                          *
*            This sample program uses no attribute fields.                 *
*            They are omitted.                                             *
*                                                                          *
*                                                                          *
****************************************************************************/
```

```
vattr.list_size = 1;                  /*Number of desired attributes  /*

return_code = LSMOV( &ctoken, msgflg, &otoken, "APPEND", "KEY",
   "VRAMKEYED", 0, &filepassword, "", &grouppassword, 0, &vattr );

emsgs( &ctoken );                     /* Retrieve messages if any     */

echeck(return_code, "LSMOV");         /* Check for error              */

file_open=1;                          /* Set to file is opened        */

/**************************************************************************
*                                                                         *
*                                Step 10                                  *
*                                                                         *
*              Write 100 80-byte records to the file                      *
*              using keys 1000 through 1099 (LSMW).                       *
*                                                                         *
**************************************************************************/

for( i = 1000; i < 1100; i++ )
{
   sprintf( buffer, "%4.4d The key for this record is %4.4d", i, i );

   for( j = 36; j < 80; j++ )
      buffer[j] = ' ';                      /* blank fill record      */

   return_code = LSMW( &otoken, buffer, sizeof(buffer), &return_rec_num );

   echeck( return_code, "LSMW" );

}

/**************************************************************************
*                                                                         *
*                                Step 11                                  *
*                                                                         *
*                    Close the VRAM file (LSMCLO).                        *
*                    Retrieve any messages (LSMMSG).                      *
*                                                                         *
**************************************************************************/

return_code = LSMCLO( &otoken, abort_flag );

emsgs( &otoken );                     /* Retrieve messages if any     */

echeck( return_code, "LSMCLO" );      /* Check for error              */

file_open = 0;                        /* Set to file NOT opened       */
```

```
/***************************************************************************
*                                                                          *
*                               Step 12                                    *
*                                                                          *
*                Open the VRAM file with mode=read and                     *
*                method=sequential (LSMOV).                                *
*                                                                          *
***************************************************************************/

return_code = LSMOV( &ctoken, msgflg, &otoken, "READ", "SEQUENTIAL",
   "VRAMKEYED", 0, &filepassword, "", &groupassword, 0, &vattr );

emsgs( &ctoken );                       /* Retrieve messages if any    */

echeck( return_code, "LSMOV" );         /* Check for error             */

file_open = 1;                          /* Set to file is opened       */

/***************************************************************************
*                                                                          *
*                               Step 13                                    *
*                                                                          *
*           Read the VRAM file until return code 5650 (EOF) is             *
*           received (LSMRS).  The program should be able to               *
*           read 100 records.                                              *
*                                                                          *
**************************************************************************/

for( i = 1000; i < 1100; i++ )
{
   sprintf( cmprbuf, "%4.4d The key for this record is %4.4d", i, i );

   for( j = 36; j < 80; j++ )
      cmprbuf[j] = ' ';           /* blank fill record                 */

   return_code = LSMRS( &otoken, buffer, sizeof(buffer),
      &return_rec_len, &return_rec_num);

   echeck( return_code, "LSMRS" );

   if( return_rec_len != 80 )
      printf( "returned record length is incorrect.\n" );

   if( strncmp( buffer, cmprbuf, sizeof(buffer) ) != 0 )
      printf( "record data mismatch for record %d.\n" );
}

return_code = LSMRS( &otoken, buffer, sizeof(buffer),
   &return_rec_len, &return_rec_num);

if( return_code != 5650 )
   printf( "Expected EOF did not occur.\n" );
```

```
/***************************************************************************
*                                                                         *
*                               Step 14                                   *
*                                                                         *
*                     Close the VRAM file (LSMCLO).                       *
*                     Retrieve any messages (LSMMSG).                     *
*                                                                         *
***************************************************************************/

return_code = LSMCLO( &otoken, abort_flag );

emsgs( &otoken );                      /* Retrieve messages if any     */

echeck( return_code, "LSMCLO" );       /* Check for error              */

file_open = 0;                         /* Set to file NOT opened       */

/***************************************************************************
*                                                                         *
*                               Step 15                                   *
*                                                                         *
*            Open the VRAM file with mode=update and method=all           *
*            (LSMOV).                                                     *
*                                                                         *
***************************************************************************/

return_code = LSMOV( &ctoken, msgflg, &otoken, "UPDATE", "ALL",
   "VRAMKEYED", 0, &filepassword, "", &grouppassword, 0, &vattr );

emsgs( &ctoken );                      /* Retrieve messages if any     */

echeck( return_code, "LSMOV" );        /* Check for error              */

file_open = 1;                         /* Set to file is opened        */

/***************************************************************************
*                                                                         *
*                               Step 16                                   *
*                                                                         *
*            Read the VRAM file in reverse key order (LSMRK).             *
*            The program should be able to read 100 records.              *
*                                                                         *
***************************************************************************/

for( i = 1099; i > 999; i-- )
{
   sprintf( cmprbuf, "%4.4d The key for this record is %4.4d", i, i );

   for( j = 36; j < 80; j++ )
      cmprbuf[j] = ' ';                /* blank fill record            */

   return_code = LSMRK( &otoken, buffer, sizeof(buffer), &return_rec_len,
      "KEY1", cmprbuf, 4, &return_rec_num);
   echeck( return_code, "LSMRK" );

/*
 * Since records were written in key order, record number 1
 * should be key 1000, and record number 100 should be key 1099.
 */

   if( return_rec_num != (i - 999) )
      printf( "returned record number is incorrect.\n" );

   if( return_rec_len != 80 )
      printf( "returned record length is incorrect.\n" );

   if( strncmp( buffer, cmprbuf, sizeof(buffer) ) != 0 )
      printf( "record data mismatch for record %d.\n", i );
}

/*
 * LSMRK should return error code 2587 because there is
 * no record with key ABCD.
 */

return_code = LSMRK( &otoken, buffer, sizeof(buffer), &return_rec_len,
   "KEY1", "ABCD", 4, &return_rec_num);

if( return_code != 2587 )
   printf( "Expected return code 2587, but received %d.\n",
   return_code );

/***************************************************************************
*                                                                          *
*                                Step 17                                   *
*                                                                          *
*        Read the VRAM file in reverse record number order (LSMRR).        *
*        The program should be able to read 100 records.                   *
*                                                                          *
***************************************************************************/

for( i = 100; i > 0; i-- )
{

/*
 * Since records were written in key order, record number 1
 * should be key 1000, and record number 100 should be
 * key 1099.
 */

   sprintf( cmprbuf, "%4.4d The key for this record is %4.4d",
      i + 999, i + 999 );

   for( j = 36; j < 80; j++ )
      cmprbuf[j] = ' ';                      /* blank fill record      */
   return_code = LSMRR( &otoken, buffer, sizeof(buffer),
      &return_rec_len, i);

   echeck( return_code, "LSMRR" );

   if( return_rec_len != 80 )
      printf( "returned record length is incorrect.\n" );

   if( strncmp( buffer, cmprbuf, sizeof(buffer) ) != 0 )
      printf( "record data mismatch for record %d.\n", i );
}

/*
 * The next LSMRR should return error code 2587,
 * indicating an invalid record number.  Record number
 * 999 does not exist.
 */

return_code = LSMRR( &otoken, buffer, sizeof(buffer),
   &return_rec_len, 999);

if( return_code != 2587 )
   printf( "Expected return code 2587, but received %d.\n",
   return_code );

/*****************************************************************************
*                                                                            *
*                                 Step 18                                    *
*                                                                            *
*           Read-keyed using key 1050 (LSMRK). Issue a                       *
*           read-next-key (LSMRNK) to read the record with key 1051.         *
*                                                                            *
*****************************************************************************/

return_code = LSMRK( &otoken, buffer, sizeof(buffer), &return_rec_len,
   "KEY1", "1050", 4, &return_rec_num);

echeck( return_code, "LSMRK" );

return_code = LSMRNK( &otoken, buffer, sizeof(buffer), &return_rec_len,
   &return_rec_num);

echeck( return_code, "LSMRNK" );

if( strncmp( buffer, "1051", 4 ) != 0 )
   printf( "LSMRNK did not return record with key 1051.\n" );
```

```
/**************************************************************************
*                                                                         *
*                               Step 19                                   *
*                                                                         *
*                  Read-keyed using key 1011 (LSMRK).                     *
*                  Then change the key to 9999 (LSMCH).                   *
*                                                                         *
**************************************************************************/

return_code = LSMRK( &otoken, buffer, sizeof(buffer), &return_rec_len,
   "KEY1", "1011", 4, &return_rec_num);

echeck( return_code, "LSMRK" );

strncpy( buffer, "9999", 4 );

return_code = LSMCH( &otoken, buffer, sizeof(buffer));

echeck( return_code, "LSMCH" );

/**************************************************************************
*                                                                         *
*                               Step 20                                   *
*                                                                         *
*                 Read-keyed using key 9999 (LSMRK).                      *
*                 Then delete the record with key 9999 (LSMDEL).          *
*                                                                         *
**************************************************************************/

return_code = LSMRK( &otoken, buffer, sizeof(buffer), &return_rec_len,
   "KEY1", "9999", 4, &return_rec_num);

echeck( return_code, "LSMRK" );

return_code = LSMDEL( &otoken);

echeck( return_code, "LSMDEL" );

/**************************************************************************
*                                                                         *
*                               Step 21                                   *
*                                                                         *
*           Read-keyed using key 9999 (LSMRK). The resulting              *
*           return code indicates that the record with key 9999           *
*           has been deleted.                                             *
*                                                                         *
**************************************************************************/

return_code = LSMRK( &otoken, buffer, sizeof(buffer), &return_rec_len,
   "KEY1", "9999", 4, &return_rec_num);

if( return_code != 2588 )
   printf( "Expected return code 2588, but received %d.\n",
   return_code );
```

```
/*************************************************************************
*                                                                       *
*                               Step 22                                 *
*                                                                       *
*                    Close the VRAM file (LSMCLO).                      *
*                    Retrieve any messages (LSMMSG).                    *
*                                                                       *
*************************************************************************/

return_code = LSMCLO( &otoken, abort_flag );

emsgs( &otoken );                       /* Retrieve messages if any     */

echeck( return_code, "LSMCLO" );        /* Check for error              */

file_open = 0;                          /* Set to file NOT opened       */

/*************************************************************************
*                                                                       *
*                               Step 23                                 *
*                                                                       *
*            Disconnect from StorHouse (LSMDIS).                        *
*            Retrieve any messages (LSMMSG).                            *
*                                                                       *
*************************************************************************/

return_code = LSMDIS( &ctoken);

emsgs( &ctoken );                       /* Retrieve messages if any     */

echeck( return_code, "LSMDIS" );        /* Check for error              */

exit();

}

/*************************************************************************
*                                                                       *
*                               EMSGS Routine                           *
*                                                                       *
*            Retrieve all error and informational messages              *
*            (LSMMSG).  Continue until all messages have been read.     *
*                                                                       *
*************************************************************************/

static void emsgs( token )

struct LSMS_TOKEN *token;

{
   long bufsize;
   long msglen;
   long return_code;
   char buffer[133];

   bufsize = sizeof(buffer);

   return_code = LSMMSG( token, buffer, bufsize, &msglen );

   while (return_code == 0)
   {
      printf("%s\n", buffer);
      return_code = LSMMSG( token, buffer, bufsize, &msglen );
   }

   return;

}

/***************************************************************************
*                                                                         *
*                                ECHECK Routine                           *
*                                                                         *
*           If the return code is not equal to zero, then                 *
*           print the return code, close the file, disconnect             *
*           from StorHouse, and exit.                                     *
*                                                                         *
***************************************************************************/

static void echeck( return_code, func )

long return_code;

char *func;

{

   long abort_flag;                    /* close argument              */

   if( return_code == 0 )              /* Return if no error          */
      return;

   printf( " $$$$$ Bad return return_code = %d. From %s.\n",
      return_code, func );

/*
 * If the StorHouse file is open, close it (LSMCLO).
 */

   if( file_open == 1 )
   {

      abort_flag = LSMF_ABORT;         /* Perform a normal close      */

      LSMCLO( &otoken, abort_flag );   /* Ignore return code          */
      emsgs( &otoken );                 /* Retrieve messages if any    */
   }

/*
 * Disconnect from StorHouse (LSMDIS).
 */

   LSMDIS( &ctoken);                    /* Ignore return code          */

   emsgs( &ctoken );                    /* Retrieve messages if any    */

   printf( "\n\n***** abnormal termination *****\n\n" );

   exit() ;
}
```

Appendix

A

Checkpoint/Restart and Programming Guidelines

This appendix contains additional technical information about programming with the Callable Interface. Information is presented in two sections:

- Checkpoint/Restart
- Programming Guidelines.

The purpose of these sections is to provide additional examples and programming tips.

Checkpoint/Restart

Checkpoints can be issued only during append operations to VRAM files. That is, the file must have been opened using either LSMOV and a mode of APPEND or LSMCO. A successful LSMCP guarantees that all data written up to the time of the checkpoint has been received and processed by StorHouse.

Only the current (most recent) revision of a file version, either accessible or software disabled, can be opened at a checkpoint. Opening a file at a checkpoint is referred to as a *restart*.

Examples

This section contains four examples that open VRAM files and issue checkpoints. The examples assume that the current version of the VRAM file DATAFILE has three revisions (see Table A-1). Revisions 1 and 2 contain no checkpoints. Revision 3 contains three checkpoints, which are referenced here as checkpoints a, b, and c.

Note (Actual checkpoints are binary numbers, not alphanumeric characters. The caller should keep track of checkpoint numbers and make no assumptions about their value.)

Table A-1: DATAFILE Revisions

Revision Number	Checkpoint	Open
1	None	LSMOV, any MODE
2	None	LSMOV, any MODE
3	a,b,c	LSMOV, any MODE or LSMOV, mode=append at any checkpoint

Revisions 1 and 2 can be opened in any mode. Revision 3 can be opened without supplying a checkpoint in any mode or in mode=append at checkpoint a, b, or c.

Example 1

The caller opens Revision 3 shown above with mode=append at checkpoint a and issues LSMCP and LSMCLO. The resulting revisions and their checkpoints are:

	Revision Number	Checkpoint
LSMOV mode=append	1	None
LSMCP	2	None
LSMCLO	3	a,d

Checkpoints b and c in the original Revision 3 are no longer accessible. The last checkpoint in the current Revision 3 is checkpoint d.

Example 2

The caller opens Revision 3, generated in Example 1, with mode=update and issues LSMCH, LSMDEL, and LSMCLO. The resulting revisions are:

	Revision Number	Checkpoint
LSMOV mode=update	1	None
LSMCH	2	None
LSMDEL	3	None
LSMCLO	4	None

There are now four revisions. Any previous checkpoints are no longer accessible. Checkpoints are accessible only in the current revision.

Example 3

The caller opens Revision 3 (generated in Example 2), using mode=append and issues LSMW, LSMCP, LSMW, LSMCP, and LSMAB. The resulting revisions are:

	Revision Number	Checkpoint
LSMOV	1	None
mode=append	2	None
LSMW LSMCP	3	None
LSMW LSMCP	4	None
LSMCLO with abort flag set	5	a, b (software disabled)

There are now five revisions. Revision 5 has two checkpoints, a and b, and is marked as software disabled because of the LSMAB.

If the caller opens Revision 5 in mode=append and supplies a checkpoint of 0 or no checkpoint number, StorHouse returns a status code of 2630 and the last checkpoint number, in this case checkpoint b.

Example 4

The caller opens Revision 4 (from Example 3) with mode=append and issues LSMW and LSMCLO. The resulting revisions are:

	Revision Number	Checkpoint
LSMOV	1	None
mode=append	2	None
LSMW	3	None
LSMCLO	4	None
	5	None

In this example, the user opened an older, accessible revision of the file to *roll back* the current revision, which was software disabled. A new Revision 5 containing the appended data now supersedes the software disabled Revision 5 from Example 3.

Programming Guidelines

The guidelines in this section apply to programs that use:

- LSMOS
- LSMOV with the StorHouse system parameter VRAM_FILE_OPEN set to true, and any mode and access method
- LSMOV with VRAM_FILE_OPEN set to false, a mode of READ, and an access method of SEQUENTIAL
- LSMOV with VRAM_FILE_OPEN set to false and a mode of APPEND or UPDATE
- LSMCO.

A program using one or more of the types of access listed above will never run to completion if the program attempts to have open at the same time files that require use of the same resource.

Defining Resources

Resources include:

- Optical volumes (for write)
- Tape volumes (for write)
- Optical disk drives (ODUs)
- Tape drives
- Transfer Manager processes.

The system parameter XFR_COUNT limits the number of Storage Manager processes.

The following situations require use of the same resource:

- Attempting to have files open on more level L volumes than available level L drives
- Attempting to have open for write two or more files that are on the same optical or tape volume
- Attempting to have open more files than the value of XFR_COUNT.

Examples

The two examples in Table A-2 illustrate what can happen when open statements require the use of the same resource. Both examples assume that:

- There are two available optical disk drives.
- All files reside on different optical disks.
- Files are opened using LSMOS with a mode of READ or LSMOV with a mode of READ and an access method of SEQUENTIAL:

Table A-2: Examples of Open Statements Resource Usage

Example 1	Example 2
OPEN FILE1	OPEN FILE1
READ FILE1	OPEN FILE2
CLOSE FILE1	OPEN FILE3
OPEN FILE2	READ FILE1
READ FILE2	READ FILE2
CLOSE FILE2	READ FILE3
OPEN FILE3	CLOSE FILE1
READ FILE3	CLOSE FILE2
CLOSE FILE3	CLOSE FILE3

Example 1 executes successfully because an ODU is always available to satisfy each LSMOS request. Because the close statement for each file releases an ODU, there are no conflicts for shared resources.

In contrast, Example 2 will not run to completion. It attempts to have three level L files open at the same time when there are only two available optical disk drives. Example 2 will wait indefinitely for an available ODU to satisfy the OPEN FILE3 request. In Example 2, contention for the optical drive is causing the problem.

User Guidelines

Applications and files should be set up to avoid resource conflicts.

- Do not plan to write to files that are in the same volume set at the same time.
- If you must read files concurrently, ensure that there are enough optical or tape drives configured in the system to handle the read requests. If there are enough drives, understand that your application may not run if a drive goes down.
- To prevent problems resulting from an insufficient number of StorHouse processes, use the interactive SHOW SYSTEM command to display the value of XFR_COUNT. If more files must be open at the same time than the value of XFR_COUNT, refer the problem to your system administrator.

Permanent Fixes

The following suggestions are *permanent fixes* to a resource conflict involving optical drives. They should not be used as a *temporary solution* for a resource conflict caused by a drive that goes down.

- To prevent problems resulting from an insufficient number of optical or tape drives when level L files that must be open at the same time reside on different optical volumes, verify that there are at least as many optical drives available in the selected library device as level L files. If there are not enough available optical drives, RELOCATE, or move, some of the level L files to a level F file set.

Note RELOCATE is a permanent move that deletes the source. Do not RELOCATE to level F unless you are willing to lose your original level L copy (SGI does not recommend this action).

- To prevent problems resulting from writing to files residing on the same level L volume, ensure that all level L files that must be open for write at the same time are in different volume sets. If files belong to the same volume set(s):
    - RELOCATE one or more files to a different volume set(s).
    - Write one or more files to the performance buffer rather than directly to a level L volume set. In other words, use VTF=NEXT.

Appendix

B

CREATE FILE Command

The StorHouse Command Language CREATE FILE command creates a new VRAM file or file version. This is functionally equivalent to using LSMCO without writing any data. The CREATE FILE command must be used to create files with new key definitions, as LSMCO cannot be used to do this.

The CREATE FILE command can be sent to StorHouse using LSMSCI. Note that your program will become less portable; some mainframe implementations do not allow CREATE FILE to be submitted using LSMSCI, because some mainframe security processing may be bypassed.

CREATE FILE

The CREATE FILE command creates a new VRAM file or file version.

Format

CREATE FILE filename

COMMAND FORMAT SUMMARY					
COMMAND, PARAMETER, OR MODIFIER	**REQUIRED COMMAND PRIVILEGE**	**REQUIRED GROUP ACCESS**	**REQUIRED FILE ACCESS**	**MINIMUM ACCOUNT ACCESS**	**DEFAULT**
CREATE FILE	RECORD	-	-	-	-
/REPORT	-	-	-	-	-
filename	-	W	-	-	(Required)
/ASCII	-	-	-	-	Binary format
/ATF=...	ATF	D	-	-	See text
/CACHE=...	-	-	-	-	/CACHE=0

COMMAND FORMAT SUMMARY					
/DIRECT	-	-	-	-	-
/EDC	-	-	-	-	See text
/EXTERNAL	-	-	-	-	No external keys
/FSET=...	-	-	-	-	Current default
/GROUP=...	SETGROUP	W	-	-	Current default
/LIMIT=...	DELETE	D	-	-	See text
/NEWPASSWORDS=...	PASSWORD	D	D	-	See text
/PASSWORDS=...	-	-	-	-	-
/REPLACE=...	DELETE	D	D	-	-
/RETENTION=...	-	-	-	-	DEFAULT
/SIZE=...	-	-	-	-	(Required)
/TYPE=...	-	-	-	-	/TYPE=RECORD
/VSET=...	-	-	-	-	Current default
/VTF=...	VTF	D	-	-	See text

Description

CREATE FILE creates the specified file or a new version of the file in StorHouse. If you specify KEYED or KEYSEQUENTIAL as the value of /TYPE, StorHouse asks you to enter key definitions. This procedure is described in the section “Key-Definition Mode,” presented later in this command description.

The /SIZE modifier is required. It specifies the maximum number of bytes needed to store the largest extent set created in an append operation. An *extent set* consists of a data extent, a DF extent, and for KEYED files, a K extent. If a file is not checkpointed, it has one extent set created from OPEN to CLOSE. If a file is checkpointed, a new extent set is created each time a checkpoint is issued.

If the file is not checkpointed, the amount of space specified by /SIZE is allocated to store the extent set that is written from OPEN to CLOSE. If the file is checkpointed, the amount of space specified by /SIZE is allocated when the file is opened and *each time* a CHECKPOINT is taken.

The section “Estimating a Value for /SIZE” is presented later in this command description to help you determine a suitable size.

When you write records into the file, the system places the records in the performance buffer and allows the backup function to copy the records to their primary file set (the file set specified by /VSET and /FSET) unless the file is created in a level F volume set and file set. If it is created in a level F volume set, the performance buffer is not used. If you specify /DIRECT, the system transfers the records directly to the primary file set, bypassing the performance buffer. The system writes file updates to the file set's update space, if available.

Parameters

filename

Specifies the StorHouse name of the VRAM file version to be created.

- FORMAT: filename

  File names must be unique within an access group. The StorHouse file name must contain 1 to 56 printable ASCII characters. At least one character must be non-blank. StorHouse translates lowercase letters to uppercase letters and compresses multiple consecutive spaces to a single space unless they are enclosed in quotes. File names containing special characters (defined in Chapter 2, "Using StorHouse Command Language") must be enclosed in quotes, unless the characters are any of the following:

  [ ] : $. ; _

  You can use the quote symbol (") in the file name as long as you enclose the name in quotes and you place two quotes (" ") wherever a single quote (") is to appear.

- DEFAULT: None; you must specify this parameter.
- ACCESS REQUIREMENTS: Write access to the file's group.

Command Modifier

/REPORT

Controls the generation of a special text response for the completion of the command. The text includes the file identifier (fid) of the new file or file version created.

/REPORT instructs StorHouse to generate a text response. /NOREPORT instructs StorHouse not to generate a text response.

- FORMAT: /REPORT or /NOREPORT
- DEFAULT: /NOREPORT

Parameter Modifiers

/ASCII

Causes the host interface to translate a host file's data from the host's native character set into ASCII characters. The host interface formats the translated data into a transportable ASCII character-stream file format while transferring the file to StorHouse.

- FORMAT: /ASCII
- DEFAULT: If you omit /ASCII, StorHouse formats the file data into transportable binary bit-stream format.
- RESTRICTIONS: Do not specify /ASCII with a different format indicator, such as /BINARY, or with files that cannot be translated into ASCII characters.
- HOST DEPENDENCIES: The native UNIX character set is ASCII, so no character translation is necessary.

  The IBM MVS Callable Interface translates data between EBCDIC characters and ASCII characters.

/ATF Specifies a value for the ATF (Access Time Factor) attribute for a file version. The ATF attribute indicates the importance of access time for the file. Setting an ATF value does not initiate a file transfer directly, but it may cause the file to be migrated in a subsequent migration.

- FORMAT: /ATF={1,2,3}

  A value of 1 indicates that a short access time for the file is very important; 2 indicates that access time is moderately important; 3 indicates that it is minimally important. The MIGRATE function migrates files off of the performance buffer, beginning with files with the largest ATF values.
- DEFAULT: If you omit /ATF when creating a new file version, the default is the current value of the ATF system parameter.

/CACHE Specifies the number of sequential records that VRAM will cache for a READ-SEQUENTIAL, READ-RECORD, or READ-KEYED function for this file when it is opened with an access mode of READ or UPDATE and an access method of RECORD and/or KEYED. The system can use this cache to optimize subsequent reads. VRAM caches n-1 records preceding the current record through n records following the current record; in other words, VRAM caches twice the specified number of records including the current record. (This assumes that there is enough cache memory available to accommodate the total number of records.)

- FORMAT: /CACHE=number_of_records

  The number of records can range from 0 to 65,535; however, StorHouse limits the number of records it caches to a number less than or equal to the number of bytes specified by the VRAM_CACHE_MAX system parameter.

Note that VRAM caches records preceding the current record up to and including the current record only if the records already reside in memory and begin in the currently loaded frame.

- DEFAULT: If you omit /CACHE, the default value is /CACHE=0 (no cache).

/DIRECT Indicates that when you write data records to the file, the system is to write the records directly to the primary file set (specified by /VSET and /FSET) specified in this command.

- FORMAT: /DIRECT
- DEFAULT: If you omit /DIRECT, the system writes the records to the performance buffer and allows the backup function to copy the data to the specified primary file set.
- PRIVILEGE: None.
- RESTRICTIONS: This modifier has the same function as /VTF=DIRECT. If you specify /DIRECT, the system ignores /VTF.

/EDC Controls the generation of error detection codes by the host interface during data transfer to StorHouse.

- FORMAT: /EDC or /NOEDC

  If you specify /EDC when executing CREATE FILE, the host interface will generate or check error detection codes when the file is transferred to or from StorHouse. If you specify /NOEDC, the host interface will not generate or check error detection codes.

- DEFAULT: If you omit /EDC, the default is given by the EDC system parameter. If the value of the parameter is TRUE, the default is /EDC. If the value of the parameter is FALSE, the default is /NOEDC.

/EXTERNAL Indicates that you will define external keys—key data provided in special records that are not a part of the file's data records. See the section, “Key-Definition Mode,” presented later in this command description.

Note: External key values cannot be changed if file records are updated. Also, records with duplicate external key values cannot be distinguished unless the record data contains information that you can use for this purpose.

- FORMAT: /EXTERNAL
- DEFAULT: If you do not specify /EXTERNAL for a KEYED file, the system assumes that the keys will be internal to the user data records.
- RESTRICTIONS: /EXTERNAL is valid only if you also specify /TYPE=KEYED.

/FSET Specifies the primary file set for the file. The performance buffer file set is not allowed. The specified file set must exist.

If /FSET and /VSET specify or default to a level F file set and volume set, StorHouse does not use the performance buffer when you write data into the file. Data is written directly to the primary file set and volume set on level F, regardless of the value of the /VTF or /DIRECT modifier, if specified.

- FORMAT: /FSET=fset_name
- RESTRICTIONS:
  - You cannot use a wild card.
  - Do not specify the performance buffer file set name.

/GROUP Specifies a file access group name and, optionally, group passwords. The specified group must exist.

- FORMAT:
  - /GROUP=groupname<::writepw>
  - /GROUP=groupname
- FORMAT RESTRICTIONS: Wild cards are not allowed in the group name.
- ACCESS REQUIREMENTS: You must have write access to the group. Also, you must specify the group's write password unless:
  - The group is not protected by a write password.
  - Your privilege bypasses write access password checks.
  - Your default access to the group includes write access.

  If you enter parameter modifiers that require delete access to the group, you must also have delete access to the group.
- DEFAULT:
  - If you omit /GROUP, the default is your current default group and default access rights.
  - If you specify the current default group name and omit the write password, the defaults for your group access rights apply.
  - If you specify a group name that is not the current default group and omit the write password, the write password defaults to null.
- PRIVILEGE: SETGROUP is required to specify any group except your default group.

/LIMIT

Specifies a value for the LIMIT attribute for a file.

- FORMAT: /LIMIT=maximum_versions

  The value of maximum_versions can range from 1 through 32768.

- DEFAULT: If you omit /LIMIT and this is a new file, the default limit is specified by the LIMIT system parameter. If this is a new version of an existing file, the default is the limit for the existing file.

- PRIVILEGE: DELETE privilege.

/NEWPASSWORDS

Assigns file access passwords to the file.

Note: You cannot obtain access to the file by specifying /NEWPASSWORDS (see the next modifier /PASSWORDS).

- FORMAT:

  - /NEWPASSWORDS=<readpw>:<writepw>:<deletepw>
  - /NEWPASSWORDS=<readpw>:<writepw>
  - /NEWPASSWORDS=readpw
  - /NONEWPASSWORDS

  Specifying /NONEWPASSWORDS is equivalent to specifying /NEWPASSWORDS with null read, write, and delete passwords.

  Passwords must be null or contain 1 to 8 characters, consisting of the following ASCII characters: A-Z (uppercase), 0-9, _ (underscore), and $ (dollar sign). StorHouse always translates passwords to uppercase characters, even if they are enclosed in quotes.

- DEFAULT: If you do not specify /NEWPASSWORDS and the file already exists, the system retains the existing passwords, if any. If you do not specify /NEWPASSWORDS and the file does not exist, the system assigns null passwords to it.

  If you specify /NEWPASSWORDS but do not specify one or more passwords, the system assigns a null password for each unspecified password.

- ACCESS REQUIREMENTS: You must have delete access to the file and group and PASSWORD privileges.

/PASSWORDS

Specifies passwords to gain access to an existing file protected by passwords. Specify a delete password to change file attributes. Specify a write password to create a new version of an existing file. If your privilege bypasses the access password checks, you do not have to specify a password.

If the file does not already exist, StorHouse ignores /PASSWORDS.

/PASSWORDS must be used in conjunction with /REPLACE.

- FORMAT:
  - /PASSWORDS=<readpw>:<writepw>:<deletepw>
  - /PASSWORDS=<readpw>:writepw
  - /PASSWORDS=readpw

  A file password can be null or contain 1 to 8 characters, and can consist of the following characters: A-Z (uppercase), 0-9, _ (underscore), and $ (dollar sign). StorHouse always translates passwords to uppercase characters, even if they are enclosed in quotes.

- DEFAULT: If you omit /PASSWORDS, the passwords default to nulls.

/REPLACE Indicates that after creating a new version of the file, the system will delete all older versions. If no file of the same name exists, StorHouse ignores this modifier. If a file of the same name exists, StorHouse verifies that you have the required access to the file and that the file is not retained before deleting it. If the existing file is retained, the CREATE FILE/REPLACE operation fails.

After StorHouse deletes the file, it does not retain the old passwords and file attributes. You must specify new passwords or attributes on the CREATE FILE statement.

- FORMAT: /REPLACE

- DEFAULT: If you omit /REPLACE and a file of the same name already exists in the directory, the system does not delete the existing version.

- PRIVILEGE: DELETE privilege

- ACCESS REQUIREMENTS: If a file with the same group and file names already exists, you must obtain delete access to the group and file.

/RETENTION Specifies the retention attribute (retention period) for the file being created.

- FORMAT:

Option	Description
/RETENTION=DEFAULT	Sets the retention period to the default value.
/RETENTION=number_of_days	Sets the retention period to the specified number of days. The retention period ends when the current date is beyond the file's last modification date plus the specified retention value. A value of 0 indicates no retention period (same as specifying ZERO). Example: /RETENTION=3 In this example, the retention period is 3 days. The retention period for a file that was last modified at 11 p.m. on December 12 expires at 11 p.m. on December 15.
/RETENTION=ZERO	Sets no retention period, which indicates the file may be deleted.
/RETENTION=FOREVER	Sets an infinite retention period, which indicates the file may never be deleted.

- DEFAULT: If you omit /RETENTION or specify /RETENTION=DEFAULT, StorHouse determines the file's default retention attribute as follows:
  - If the file's resident file set has a retention attribute equal to FOREVER, ZERO, or a specified number of days, the file set retention attribute determines the default file retention attribute.
  - If the file's resident file set has a retention attribute of DEFAULT, the RETENTION_MODE system parameter determines the default file retention attribute. If RETENTION_MODE is set to BASIC, the default file retention is ZERO. If RETENTION_MODE is set to STRICT, the default file retention is FOREVER.
- RESTRICTIONS: None.

/SIZE Specifies the number of bytes of storage space to allocate for the file whenever the file is opened for an append operation and whenever a checkpoint is issued. The value must contain enough space for the largest extent set that is written. This extent set includes a data extent, a DF extent, and for KEYED files, a K extent.

- If the file is not checkpointed, /SIZE specifies the space required to store the extent set that is written from OPEN to CLOSE.

- If the file is checkpointed, the space specified by /SIZE is allocated when the file is opened and *each time* a CHECKPOINT is taken.

Note: The largest value that you can specify for /SIZE is 2G. Do not assign a value for /SIZE greater than the size of the volume (or performance buffer) that will contain the extent set. Otherwise, CREATE FILE returns an error.

If the system cannot allocate enough space when the file is opened, it rejects the open. If the system cannot allocate enough space after a checkpoint, it rejects the next write. After a file has been closed or checkpointed, StorHouse returns any unused file storage space to the file set as free space.

For updates, StorHouse does not use the /SIZE value. It automatically allocates space for each extent separately. It allocates as much space as required, up to the file set LIMIT.

Caution: After you have used CREATE FILE to create the file, you cannot change the value of /SIZE.

See “Estimating a Value for /SIZE” for informaton about estimating values for /SIZE.

- FORMAT: /SIZE=number_of_bytes

  The number_of_bytes value can be specified as n, nK, nM, or nG.

  The letter n represents a numeric field. The value of n can range from 0 up to 2000000000. (Do not include commas when specifying a number with more than three digits.) K indicates that the number is in 1,000-byte units; M indicates 1,000,000-byte units; and G indicates 1,000,000,000-byte units. If K, M, or G is not present, the number defaults to 1-byte units.

  The maximum value of /SIZE is limited by the capacity of the volume to which the file is to be written. If you do not use /DIRECT or /VTF=DIRECT, the value must be smaller than the capacity of the largest partition of the performance buffer.
- DEFAULT: None; you must specify this modifier.

/TYPE Specifies the type of file organization desired. A /TYPE value of RECORD indicates that the file can only be accessed sequentially or by record number. KEYED or KEYSEQUENTIAL indicates that the file can be accessed sequentially, by record number, or by key.

KEYSEQUENTIAL files are like KEYED files, but with the following restrictions:

- You can define only one key.

- Duplicate key values are not allowed.

- When writing records into the file, you must write the records in ascending key value order.

- When updating a record, you cannot change the key value.

If /TYPE is KEYED or KEYSEQUENTIAL, StorHouse requests you to enter key definitions. For further information about key definitions and an explanation of how to enter keys, see the section "Key-Definition Mode."

- FORMAT:
  - /TYPE=KEYSEQUENTIAL
  - /TYPE=KEYED
  - /TYPE=RECORD

- DEFAULT: /TYPE=RECORD

- RESTRICTIONS: The VRAM_KEYED system parameter must be set to TRUE for VRAM_KEYED files to be created.

/VSET Specifies the file's primary volume set.

If /VSET and /FSET specify or default to a level F volume set and file set, StorHouse does not use the performance buffer when you write data into the file. Data is written directly to the primary file set and volume set on level F, regardless of the value of the /VTF or /DIRECT modifier, if specified.

- FORMAT: /VSET=vset_name

- DEFAULT: If you omit /VSET, the default is your current default volume set.

- RESTRICTIONS:
  - Wild cards are not allowed.
  - The volume set must be a primary volume set.

/VTF Specifies a file version's Vulnerability Time Factor (VTF) attribute, which determines how long StorHouse can leave new extents of the file version in the performance buffer before copying them to their primary file set.

- FORMAT: /VTF=NEXT, /VTF=NOW, or /VTF=DIRECT

  - If you specify /VTF=NEXT, the file is written to the performance buffer. The next time a backup occurs, the file is copied to its primary file set.

  - If you specify /VTF=NOW, StorHouse copies the new version from the performance buffer to its primary file set as part of the command.

  - If you specify /VTF=DIRECT, the file bypasses the performance buffer. Extents are written directly to their primary file set.

- DEFAULT: If you omit /VTF, the default is the value of the VTF system parameter.
- RESTRICTIONS: If you specify /DIRECT on the command, /VTF is ignored.
- ACCESS REQUIREMENTS: Delete access to the group and VTF privilege.

Estimating a Value for /SIZE

The value of /SIZE should be based on the size of the largest extent set in the file. Each time the file is opened with a mode of APPEND or checkpointed, StorHouse allocates the amount of space specified by /SIZE.

The minimum value of /SIZE should be set to the size of the largest data extent that will be written plus the sizes of the largest DF and K extents (for KEYED files) that are created.

Note: The largest value that you can specify for /SIZE is 2G. Do not assign a value for /SIZE greater than the size of the volume (or performance buffer) that will contain the extent set. Otherwise, CREATE FILE returns an error.

During an append to a VRAM file, whenever a CHECKPOINT is issued, or the file is closed, StorHouse produces a DF extent, a data extent, and for KEYED files, a K extent. If you plan to issue CHECKPOINT for the file being created, base your /SIZE estimate on the largest extents. The latest DF and K extents are usually the largest.

Once you have executed CREATE FILE, you cannot change the value of /SIZE. Therefore, you may want to increase your estimate to avoid running out of space.

In the formulas that follow, all sizes are in units of bytes.

Estimating Data Extent Size

The size of a data extent written during an append operation for a KEYED or RECORD file is given by Formula 1, where D is the largest number of bytes of user data that will be written in one extent set, and nrecs_e is the largest number of records in one extent set:

Formula 1 data size = 32,000 + (1.006 ∗ (D + (nrecs_e ∗ 7)))

If there are no checkpoints, nrecs_e is the same as the number of records in the file.

Formula 2 gives the size of a data extent written during an append operation for a KEYSEQUENTIAL file, where K1 is the size of the key:

Formula 2 data size = 32,000 + (1.006 ∗ (D + nrecs_e ∗ (7 + K1)))

Estimating DF Extent Size

For VRAM files, DF extents normally range from a minimum of 1.2 KB (1,200 bytes) to a maximum of 10 KB (10,000 bytes) with the average being about 6 KB (6,000 bytes) or less. Files with keys, checkpoints, and/or a large number of updates have larger DF extents. In addition, whenever a checkpoint is issued during an append to a VRAM file, StorHouse produces another DF extent. The newer DF extents are usually larger because they contain additional entries.

The DF extents include various tables. Calculate the sizes of these tables, as applicable, as described in the following paragraphs.

To compute DF size, always add the minimum DF size of 1.2 KB to the sum of the estimated table sizes.

All Files

For all files, estimate the size of the data table as follows, where data_xtnts is the total number of data extents in the file:

Formula 3 data table size = 13 + (12 ∗ data_xtnts)

A new data extent is created when a file is opened in mode=APPEND and either a checkpoint is performed or the file is closed normally.

Checkpointed Files

For files that will be checkpointed, estimate the size of the checkpoint table as follows, where max_cpts is the maximum number of checkpoints in one append operation:

Formula 4 checkpoint table size = 13 + (8 ∗ max_cpts)

Files with Keys

For files with keys, estimate the size of the key name and key segment location tables. Estimate the size of the key name table as follows, where nkeys is the total number of keys defined for the file:

Formula 5 key name table size = 13 + (58 ∗ nkeys)

Estimate the size of the key segment location table as follows, where nsegs is the total number of key segments defined for the file:

Formula 6 key segment location table size = 13 + (6 ∗ nsegs)

Keysequential Files

For keysequential files, estimate the size of the key index table as follows, where K1 is the size of the key in bytes; data_xtnts is the total number of data extents in the file (a new data extent is created when the file has been opened in APPEND mode and either a checkpoint is performed or the file is closed normally); and nrecs_t is the total number of records in the file:

Formula 7 key index table size = 13 + (K1 + 6) ∗ (data_xtnts + (nrecs_t ∗ (K1 + 2) /31711))

A key index table is created only for KEYSEQUENTIAL files.

Updated Files

For files that will be updated (records changed or deleted), estimate the size of the change and record modification tables.

Note: These tables need to be included in your estimate only if you plan to write additional records after updates have been performed.

Estimate the size of the change table as follows, where chg_xtnts is the total number of change extents:

Formula 8 change table size = 13 + (8 ∗ chg_xtnts)

Change extents exist only for files created with the StorHouse system parameter VRAM_UPDATE set to 1. A new change extent is created each time a file is opened with mode=UPDATE, records are changed (not just deleted), and the file is closed normally.

Estimate the size of the record modification table as follows, where nupds is the number of update entries:

Formula 9 record modification table size = 13 + (9 ∗ nupds)

An entry represents either an update of a single record or an updated group of consecutive records (all changed or all deleted) in order during one update operation.

Estimating K Extent Size

For KEYED files, the K (key data base) extent consists of an area for key data and one index area for each user-defined key. The size of a K extent is the sum of the allocations made for all areas. The amount of data stored in each area can be estimated, but the actual amount may vary due to storage overhead, index compression, and the distribution of keys.

Note: Whenever CHECKPOINT is issued during an append to a keyed VRAM file, StorHouse produces another K extent. Your estimate for each checkpoint should be based on the size of the last (largest) K extent.

Estimate the size of the key data area as follows, where nrecs_t is the total number of records with keys that are written to the file in all extent sets, and K is the sum of the sizes of all user-defined keys:

Formula 10 key data area size = 4096 + (K + 12) ∗ nrecs_t ∗ 1.03

Estimate the size of each key index area for user-defined keys as follows, where Kn is the size of key number n:

Formula 11 Kn index area size = (Kn + 12) ∗ nrecs_t ∗ 2.06

Note: These formulas are estimates. It is advisable to increase your calculated value for /SIZE to avoid running out of space.

Example 1

The following example illustrates how to use formulas 1, 3, 5, 6, 10, and 11 to estimate the value of /SIZE for a file *without* checkpoints (one data extent). This example assumes that:

- You are creating a KEYED VRAM file with one internal key and one key segment.
- The key is 10 bytes long.
- Each record is 500 bytes long and contains a key.
- 1,000 records will be written to the file.
- No checkpoints will be issued during the APPEND to this file.

Because there are no checkpoints, there will be one extent set containing 500 KB of user data (500 * 1000).

In this example, the value of /SIZE equals the sum of the allocations for the following:

- Data extent
- DF extent, which includes the sum of the following:
    - Minimum value of 1.2 KB
    - Data table size
    - Key name table size
    - Key segment location table size.
- K extent.

Table 2-1 shows the /SIZE calculations for the preceding example.

Table 2-1: Estimating /SIZE for a File without Checkpoints

Formula	Definitions
Formula 1 Data Size = 32,000 + (1.006 x (D + (nrecs_e x 7))) = 32,000 +(1.006 x (500,000 + (1000 x 7))) = 32,000 + 510,042 = 542,042 bytes or **543KB**	D = Largest number of bytes of user data in one extent set [1] nrecs_e = number of records in one extent set [2]
Formula 3 Data Table Size = 13 + (12 x data_xtnts) = 13 + (12 x 1) = 13 + 12 = 25	data_xtnts = number of data extents
Formula 5 Key Name Table Size = 13 + (58 x nkeys) = 13 + (58 x1) = 13 + 58 = 71 bytes	nkeys = number of keys
Formula 6 Key Segment Location Table Size = 13 + (6 x nsegs) = 13 + (6 x 1) = 13 + 6 = 19	nsegs = number of key segments
DF Extent Size = 1,200 + 25 + 71 + 19 = 1,315 bytes or **2KB**	
Formula 10 Key Data Area = 4,096 + (K + 12) x nrecs_t x 1.03 = 4,096 + (10 + 12) x 1000 x 1.03 = 4,096 + 22,660 = 26,756 bytes or **27KB**	K = sum of the sizes of all user-defined keys nrecs_t = total number of records containing keys in the file
Formula 11 Key Index Area = (Kn +12) x nrecs_t x 2.06 = (10 + 12) x 100 x 2.06 = 22 x 1000 x 2.06 = 45,320 bytes or **46KB**	Kn = size of key number n nrecs_t = total number of records containing keys in the file
The value of /SIZE = 543KB + 2KB + 27KB + 46KB = **618KB**	

[1]Total number of bytes because there are no checkpoints

[2] Same as nrecs_t because there are no checkpoints

K extent size is cumulative; it grows as more records with keys are appended to the file. In the preceding example, the file was appended only once. Instead, suppose that your application appends 1,000 records to the file three times. Therefore, after the third append, the key data base contains 3,000 key values. When determining the value for /SIZE in this application, you should use 3,000 for the value of n in formulas 10 and 11.

Example 2

The following example illustrates how to use formulas 2 through 7 to estimate the value of /SIZE for a file *with* checkpoints. This example assumes that:

- You are creating a VRAM KEYSEQUENTIAL file.
- The key is 50 bytes long and contains one key segment.
- The user records are 2 KB long.
- The total size of the user data is no larger than 6 GB.
- The file will be checkpointed because it is too large to fit on available media in one extent set.
- No more than 250 MB and no less than 240 MB of user data will be written for each checkpoint.
- Because the user data is no larger than 6 GB, and the minimum amount of data for one checkpoint is 240 MB, the maximum number of checkpoints (including the final close) is 25.
- Because the maximum amount of data in one checkpoint is 250 MB and the size of one record is 2 KB, the maximum number of user records in one extent set is 125,000.
- The file will not be updated.

In this example, the value of /SIZE equals the sum of the allocations for the following:

- Largest data extent
- Largest DF extent, which includes the sum of the following:
    - Minimum value of 1.2 KB
    - Data table size
    - Checkpoint table size
    - Key name table size
    - Key segment location table size
    - Key index table size.

Table 2-2 shows the /SIZE calculations for the preceding example.

Table 2-2: Estimating /SIZE for a File with Checkpoints

Formula	Definitions
Formula 2 Data Size = 32,000 + (1.006 x (D + nrecs_e x (7 +K1))) = 32,000 + (1.006 x (250,000,000 + (125,000 x 57))) = 32,000 + 258,667,750 = 258,699,750 bytes or **258,700KB**	D = largest number of bytes of user data that is written in one extent set nrecs_e = number of records in one extent set K1 = size of key
Formula 3 Data Table Size = 13 + (12 x data_xtnts) = 13 + (12 x 25) = 13 + 300 = 313 bytes	data_xtnts = number of data extents
Formula 4 Checkpoint Table Size = 13 + (8 x max_cpts) = 13 + (8 x 25) = 13 + 200 = 213 bytes	max_cpts = maximum number of check-points
Formula 5 Key Name Table Size = 13 + (58 x nkeys) = 13 + (58 x1) = 13 + 58 = 71 bytes	nkeys = number of keys
Formula 6 Key Segment Location Table Size = 13 + (6 x nsegs) = 13 + (6 x 1) = 13 + 6 = 19 bytes	nsegs = number of key segments

Formula 7 Key Index Table Size
= 13 + (K1 + 6) x (data_xtnts + (nrecs_t x (K1 + 2)/31,711))
= 13 + (50 + 6) x (25 + (3,000,000 x (50 + 2)/31,711))
= 13 + 56 x (25 + 4,920)
= 13 + 276,920
= 276,933 bytes
	K1 = length of key
	
	data_xtnts = number of extents
	
	nrecs_t = number of records in the file

DF Extent Size = 1,200B + 313B + 213B + 71B + 19B + 276,933B =287,749B or **288 KB**	
The value of /SIZE = 258,700KB + 288KB = **258,988KB**	

Key-Definition Mode

There are two types of keys: *internal* and *external*. Internal keys are fields that are contained within each data record. External keys are fields that are located in special key records that are associated with separate data records.

Internal keys can be defined for KEYED or KEYSEQUENTIAL files. External keys can be defined for KEYED files only. However, a single KEYED file can have only one type of key: either internal or external.

When you enter a CREATE FILE command and specify /TYPE=KEYED and /EXTERNAL, StorHouse puts you in external key-definition mode with the following prompt:

```
You may now enter EXTERNAL KEY definitions.
HELP is available.
COMMAND?
```

When you enter CREATE FILE with /TYPE=KEYED (and omit /EXTERNAL) or /TYPE=KEYSEQUENTIAL, StorHouse puts you in internal key-definition mode with the following prompt:

```
You may now enter KEY definitions.
HELP is available.
COMMAND?
```

Your program must cycle through these text lines and must supply responses to the prompt.

StorHouse provides six commands to help define internal or external keys: KEY, DELETE, EXIT, HELP, INSERT, and LIST. KEY and EXIT are generally the only commands used from a program. For descriptions of the other commands, see the *Command Language Reference Manual*.

KEY Command

You must enter at least one key definition for KEYED and KEYSEQUENTIAL files. Otherwise, StorHouse does not create the VRAM file and returns an error status when you exit key-definition mode (using the EXIT command). Once you exit from key-definition mode, you cannot define more keys for the file.

You can enter a maximum of 31 internal or external keys for KEYED files but *only* one key for KEYSEQUENTIAL files. If you define more than the maximum number of keys allowed for a given file type, StorHouse returns an error message but allows you to process other commands.

Defining Keys

In the following discussion, angle brackets (< >) enclose optional entries and ellipses (…) indicate a repetition of the preceding information.

To define a key, your program must supply the following text string in response to the COMMAND? prompt:

```
KEY key_name start_position:length <start_position:length <...>>
```

A key definition is composed of three items: key name, starting-byte position of the key in the user data record (or in the external key record), and key length in bytes. A space separates key_name from start_position. A colon separates start_position from the length of the key.

Key_name identifies the name of the key. Key names must contain 1 to 56 characters and include the characters A-Z (lowercase letters are translated to uppercase), the numbers 0-9, and hyphen (-). No other special characters are allowed.

Start_position specifies the byte position in the record where the key data begins. Length indicates the number of consecutive bytes of key data that begin at the start_position. For example, 5:10 indicates that the key begins in byte position 5 and has a length of 10 bytes.

Each key has one or more starting-byte positions and lengths. Start_position must be a number from 1 (the first byte of a record) to the size of a record. Each start_position has a corresponding length. Length must be a number from 1 to 254.

After each key definition is entered, the COMMAND? prompt is received.

In the following example, only one starting byte and length are specified for the key:

```
COMMAND? KEY LAST-NAME 16:15
```

LAST-NAME is the name of the key. The key begins at byte position 16 in the record and is 15 bytes long.

More than one key can reference the same information. For example:

```
COMMAND? KEY FIRST-INITIAL 1:1
COMMAND? KEY FIRST-NAME 1:15
```

The keys FIRST-INITIAL and FIRST-NAME both start in byte-position 1 and include the first character in the data record.

Concatenating Keys

Key definitions can be concatenated by entering more than one start_position and length combination on one KEY command. Each start_position and length combination defines a key segment. StorHouse extracts key data specified by each combination and combines the extractions in the order that they are listed on the KEY command. The sum of the lengths of each segment is the total length of the key, with the maximum key length remaining 254 bytes. Start_position and length combinations can overlap.

In the following command, CLAIM-HISTORY is the key name. Six key segments are defined, starting at byte positions 86, 100, 125, 145, 195, and 220. The total key length is 65 bytes, which is the combined length of all key segments.

```
COMMAND? KEY CLAIM-HISTORY 86:3 100:6 125:9 145:22 195:10 220:15
```

If a key definition is longer than one command line, type a hyphen as the last character on the current line and press R . StorHouse prompts for the continuation with "…?".

For example:

```
COMMAND? KEY CLAIM-HISTORY 86:3 100:6 125:9 145:22 195:10 220:15 -
...? 300:100 405:6
```

Commenting KEY Commands

To comment key definitions, type an exclamation point as the first character after the COMMAND? prompt:

```
COMMAND? ! THIS KEY IS FOR LAST NAME
COMMAND? KEY LAST-NAME 16:15
```

Comments are not saved by StorHouse but may be useful for host program documentation.

Sample Key Definition Series

The following commands represent a sample key definition series:

```
COMMAND? ! Last name of client
COMMAND? KEY LAST-NAME 16:15
COMMAND? ! Street address of client
COMMAND? KEY ADDRESS 31:25
COMMAND? KEY CITY 56:15
COMMAND? KEY STATE 71:2
COMMAND? KEY CLAIM-HISTORY 86:3 100:6 125:9 145:22 195:10 220:15 -
...? 300:100 405:6
```

StorHouse assigns a number to each key. In the preceding example, LAST-NAME is key number 1, ADDRESS is key number 2, CITY is key number 3, STATE is key number 4, and CLAIM-HISTORY is key number 5.

EXIT Command

The EXIT command causes StorHouse to exit from key-definition mode. The following command terminates the key definition process, completes the CREATE FILE command, and returns you to the *command-ended* state:

```
COMMAND? EXIT
```

Examples

For illustrations of how to create VRAM files, see Sections 8 and 9 of the sample program in Chapter 6.

Installing and Using the StorHouse API for Windows

The StorHouse API software for Windows allows you to communicate with StorHouse from a user application program running on an IBM PC or compatible computer under Microsoft® Windows NT™, 2000, or XP using TCP/IP. This release of the StorHouse API works with all versions of the Win32 API.

The Windows StorHouse API consists of one Dynamic Link Library (DLL) that is called from your application. You must install the DLL in the same directory as the application that uses it or ensure that the DLL is in the application's path.

Installing the Windows StorHouse API

This section describes how to install the StorHouse API for Windows. It lists the installation requirements, installation diskette contents, and installation steps.

Installation Requirements

To install and run the StorHouse API for Windows, you must have the following:

- An SGI StorHouse system attached to a local area network with TCP/IP support
- Windows NT, 2000, or XP operating system
- TCP/IP network software
- At least 4 MB of hard disk space
- A 3.5-inch diskette drive

- The 3.5-inch SGI installation diskette containing the StorHouse API for Windows software.

Installation Diskette Contents

The installation diskette contains one file called install.exe that installs the SM.DLL, sample C source code, and import libraries for Borland® and Microsoft. The sm.dll function stubs (sm.lib files) needed to link your code are located in either the BORLANDLib or MSCLib directories. Use the directory appropriate for your compiler.

The installation creates the following directories and files:

Files and Directories	Description
\SMAPI\BORLANDLib\sm.lib	Import library for Borland projects
\SMAPI\MSCLib\sm.lib	Import library for Microsoft projects
\SMAPI\Sample\COPYRIGHT	Copyright notice
\SMAPI\Sample\readme	Release information for the build
\SMAPI\Sample\sample.c	Sample C program
\SMAPI\Sample\testthrd.c	Sample multithreaded program
\SMAPI\lsmdefs.h	Header file for functions
\SMAPI\sm.dll	StorHouse API for Windows DLL
\SMAPI\filetek.ini	Sample FILETEK.INI file
\SMAPI\readme.txt	Installation and use instructions for StorHouse API for Windows

Compiling and Linking the Sample Code With the StorHouse API for Windows

The sample code provided on the installation diskette may be compiled and linked with either the Microsoft or Borland C compilers.

Sample Programs

There are two sample programs included with the StorHouse API for Windows:

- SAMPLE.C – This is the same program that is documented in Chapter 6, “Sample Program.” It is a single-threaded application that communicates with a single StorHouse system. It exercises most of the StorHouse API for Windows functions.

- TESTTHRD.C – This is a variation of the sample.c program. It uses two threads and can access one StorHouse system in each thread. Both threads can access the same StorHouse system for testing. This program shows how to use the StorHouse API for Windows in a multi-threaded environment. It is critical that all activity for a single c-token occur in the same thread that creates the c-token. In other words, the thread that executes LSMCON to connect to StorHouse and obtain a c-token must be the thread that opens, reads, writes, closes, and executes LSMDIS with that c-token. Multiple connections can occur concurrently in different threads. Therefore, separate operations can take place at the same time, but each operation must have its own c-token. Each set of operations from LSMCON to LSMDIS using the same c-token must occur in separate threads. See the sample program for more information.

FILETEK.INI File

The filetek.ini file is required to use DLL in Windows. This file requires only one section for use with SM.DLL, formatted as follows:

```
[SMList]
sm_identifier=address,port

Below is an example:

[SMList]
smsys=100.99.98.1,1200
```

The sm_identifier must contain a maximum of six characters, as specified in the "LSMCON - Connect" section on page 5-3. The address must be the IP address of the StorHouse system or its domain name server (DNS) name. The port is 1200.

The filetek.ini file must reside in the WINDOWS directory and must contain the SMList section with an entry for each StorHouse system that you intend to access.

Using DLL

To use DLL with your program, you must create a project in Microsoft Visual C++™, Borland C++, or Borland C++ Builder™. Any program that calls a StorHouse API for Windows function must include the lsmdefs.h header file. Your project must include the Microsoft or Borland version of the sm.lib import library file.

The sm.dll file must be included in the directory where your executable file resides, or in your executable file's working directory.

Appendix

D

Installing and Using the OS/2 StorHouse API

The StorHouse API software allows you to communicate with StorHouse from a user application program running on an IBM PC or compatible computer under OS/2 on PC/TCP™ network software for OS/2.

Installing the OS/2 StorHouse API

This section tells you how to install the OS/2 StorHouse API. It lists the installation requirements and presents the installation steps.

Installation Requirements

To install and run the OS/2 StorHouse API, you must have the following:

- OS/2 operating system version 1.3 or higher
- PC/TCP network software for OS/2 (version 1.2 or higher) and one of the interface cards that it supports
- Microsoft C 6.0 compiler and linker for OS/2, or any other 16-bit compiler that generates Microsoft-compliant code for OS/2
- At least 4 MB of hard disk space for the PC/TCP runtime library
- A 3.5-inch disk drive
- The 3.5-inch SGI installation disk containing the OS/2 StorHouse API software.

Installation Disk Contents

The installation disk contains the following directories:

- \LIB – contains the host interface library for OS/2 (HOST.LIB).
- \SAMPLE – contains the following files:
  - ERR
  - ENV.CMD (an OS/2 batch file used to set the OS/2 StorHouse software communications parameters)
  - LSMDEFS.H
  - READ.ME (software release notes)
  - SAMPLE.C (a sample program)
  - SAMPLE.EXE
  - SAMPLE.MAK (a sample makefile)
  - SAMPLE.MAP
  - SAMPLE.OBJ
  - WFN.H
  - WFW.H
  - WMI.H
  - WVI.H
  - WVN.H
  - WVW.H
  - WYCN.H
  - WYI.H
  - WYW.H
  - WZI.H
  - README and README.WRI—contain the software release notes.

Installation Steps

▼ To install the OS/2 StorHouse API software:

1. Copy all of the files from the disk onto your PC.
2. Using the SET command, configure the OS/2 operating system environment so that it recognizes the locations of the SGI libraries and files.

   The sample program uses two environment variables for configuration:

   - server_name – the host's TCP/IP name
   - server_link – the host's socket port ID for TCP/IP.

Compiling and Linking an Application With the OS/2 StorHouse API

This section presents the instructions for compiling and linking the OS/2 StorHouse API software with the Microsoft C 6.0 compiler program. The sample program, SAMPLE.C, provides a model for StorHouse and PC/TCP function calls running under OS/2.

To compile and link an application using the StorHouse functions and the PC/TCP libraries, you *must* use the Microsoft C 6.0 compiler for OS/2. PC/TCP software does not guarantee that its libraries will work if the code is compiled and linked with a different compiler and/or linker.

Microsoft C 6.0 Compiler and Linker Switches

Sample compiler and linker switches for CFLAGS and LFLAGS that must be set are:

- CFLAGS = -c -AI -MT
- LFLAGS = /ST:8192/NOD/NOI

You *must* use the -MT compiler switch because the PC/TCP libraries use multithreaded OS/2 libraries. The switch tells the compiler to generate code for the multithreaded libraries.

The -D_DLL switch tells the compiler to use FTPCRT.LIB (the PC/TCP library that hooks into the C run-time DLL RTPCRT.DLL).

Linking the OS/2 StorHouse API

The directories in which the linker finds the library files may vary depending on your hard disk directory structure. However, the order in which the linker processes the libraries is very important. If you use a sequence of libraries for the linker other than what the sample makefile shown in the next section specifies, your application might not link or run properly. The sample makefile specifies the following libraries:

- SOCKET.LIB – a PC/TCP library
- HOST.LIB – the StorHouse host software library
- FTPCRT.LIB – a PC/TCP library
- OS2.LIB – a Microsoft OS/2 library
- LLIBCMT.LIB – a Microsoft OS/2 library

Note: Be sure to specify LLIBCMT.LIB as the OS/2 multithreaded library.

Compiling and Linking Method

The preferred method of compiling and linking SAMPLE.C (or any other application using StorHouse and the PC/TCP libraries) is presented in the following sample makefile. This method enables structure packing and a higher level of warnings to be set. For more information on compiler and linker flags, see your Microsoft C 6.0 programmer's manual.

Sample Makefile

The following makefile is a sample of the commands you should use to compile and link a program:

```
CFLAGS = -cW3 -Alfu -Zlp -G2 -J -Lp -MT -D_DLL-D_MT
LFLAGS = /ST:8192  /SE:2048 /A:16 /NOD /NOI PMTYPE:VIO

sample.exe:  sample.obj
                     link $(LFLAGS) sample, sample,, socket
host os2 ftpcrt llibcmt;

sample.obj:  sample.c
                     cl $(CFLAGS)  sample.c
```

Index

NOTE: This index lists all special characters, such as ! and /, and numeric characters before alphabetic characters.

Symbols

/ASCII parameter modifier
  CREATE FILE command B-4

/ATF parameter modifier
  CREATE FILE command B-4

/CACHE parameter modifier B-4

/DIRECT parameter modifier B-5

/EDC parameter modifier
  CREATE FILE command B-5

/EXTERNAL parameter modifier B-5

/FSET parameter modifier
  CREATE FILE command B-6

/GROUP parameter modifier
  CREATE FILE command B-6

/LIMIT parameter modifier
  CREATE FILE command B-7

/NEWPASSWORDS parameter modifier
  CREATE FILE command B-7

/PASSWORDS parameter modifier
  CREATE FILE command B-7

/REPLACE parameter modifier
  CREATE FILE command B-8

/REPORT command modifier
  CREATE FILE command B-3

/RETENTION parameter modifier
  CREATE FILE command B-9

/SIZE parameter modifier
  CREATE FILE command B-9
  estimating total value B-12

/TYPE parameter modifier
  CREATE FILE command B-10

/VSET parameter modifier
  CREATE FILE command B-11

/VTF parameter modifier
  CREATE FILE command B-11

A

access privileges for account 2-2

access_method argument 5-21

account (StorHouse)
  access privileges 2-2
  account identification code (AID) 2-1
  command privileges 2-2
  default access group 2-2
  default access rights 2-2
  definition 2-1
  passwords 2-2
  privileges 2-2

account argument 5-3

account identification code (AID) 2-1

account password (StorHouse) 2-2

any-token argument
  LSMAB general usage function 5-61
  LSMASY general usage function 5-57
  LSMMSG general usage function 5-59

APIs (StorHouse)
  OS/2 D-1
  Windows C-1

arguments
  access_method 5-21
  account 5-3
  any-token
    LSMAB general usage function 5-61
    LSMASY general usage function 5-57
    LSMCK general usage function 5-55
    LSMMSG general usage function 5-59
  buffer
    LSMR data transfer control function 5-31
    LSMRK data transfer control function 5-37
    LSMRNK data transfer control function 5-39
    LSMRR data transfer control function 5-35
    LSMRS data transfer control function 5-33
    LSMW data transfer control function 5-41
    LSMWK data transfer control function 5-43
  buffer_size
    LSMR data transfer control function 5-31
    LSMRK data transfer control function 5-37
    LSMRNK data transfer control function 5-39
    LSMRR data transfer control function 5-35
    LSMRS data transfer control function 5-33
  cr_buf 5-50
  c-token
    LSMCO file operation function 5-15
    LSMCON session control function 5-3
    LSMDIS session control function 5-5
    LSMOS file operation function 5-8
    LSMOV file operation function 5-21
    LSMSCI StorHouse command interface function 5-50
  file_attrib
    LSMCO file operation function 5-16
    LSMOS file operation function 5-9
    LSMOV file operation function 5-22
  file_location
    LSMCO file operation function 5-16
    LSMOS file operation function 5-9
  file_name
    LSMCO file operation function 5-15
    LSMOS file operation function 5-8
    LSMOV file operation function 5-21
  file_options 5-11
  file_password 5-15
  file_passwords
    LSMOS file operation function 5-9
    LSMOV file operation function 5-21
  group_name
    LSMCO file operation function 5-15
    LSMOS file operation function 5-9
    LSMOV file operation function 5-21
  group_password 5-15
  group_passwords
    LSMOS file operation function 5-9
    LSMOV file operation function 5-22
  key 5-43
  key_length
    LSMRK data transfer control function 5-37
    LSMWK data transfer control function 5-43
  key_name 5-37
  key_value 5-37
  message_buffer 5-59
  message_buffer_size 5-59
  message_flag
    LSMCO file operation function 5-15
    LSMCON session control function 5-3
    LSMOS file operation function 5-8
    LSMOV file operation function 5-21
  mode
    LSMOS file operation function 5-8
    LSMOV file operation function 5-21
  model_file_name 5-16
  o_token
    LSMCO file operation function 5-15
    LSMOS file operation function 5-8
    LSMOV file operation function 5-21
  o-token
    LSMCH data transfer control function 5-47
    LSMCLO file operation function 5-28
    LSMCP file operation function 5-26
    LSMDEL data transfer control function 5-46
    LSMR data transfer control function 5-31
    LSMRK data transfer control function 5-37
    LSMRNK data transfer control function 5-39
    LSMRR data transfer control function 5-35
    LSMRS data transfer control function 5-33
    LSMW data transfer control function 5-41
    LSMWK data transfer control function 5-43
  p1_value 5-57
  password 5-3
  record_length
    LSMW data transfer control function 5-41
    LSMWK data transfer control function 5-43
  rel_rec_num 5-22
  resp_buf 5-51

resp_bufsize 5-51
resp_info 5-51
return_ckpt_num 5-26
return_rec_len
LSMR data transfer control function 5-31
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33
return_rec_num
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33
LSMW data transfer control function 5-41
LSMWK data transfer control function 5-43
returned_message_len 5-59
revision 5-21
set_flag 5-57
sm_identifier 5-4
subsystem_identifier 5-4
version 5-9

ASCII characters, printable 2-3

asynchronous mode 5-1

ATF command privilege
CREATE FILE filename /ATF command B-1

B

binary data representation 2-6

buffer argument
LSMR data transfer control function 5-31
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33
LSMW data transfer control function 5-41
LSMWK data transfer control function 5-43

buffer_size argument
LSMR data transfer control function 5-31
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33

C

C language 5-1

c_token session link identifier 1-2, 2-1

Callable Interface
description 1-1
function hierarchy 1-1

change table size B-14

character strings 4-1

checkpoint
issuing B-12
table size B-13

checkpoint/restart
examples A-1
guidelines A-1

checkpointed files B-13

command modifiers
/REPORT
CREATE FILE command B-3

command privileges
ATF
CREATE FILE filename /ATF command B-1
DELETE
CREATE FILE filename /LIMIT command B-2
CREATE FILE filename /REPLACE command B-2
for account 2-2
PASSWORDS
CREATE FILE filename /NEWPASSWORDS command B-2
RECORD
CREATE FILE command B-1
SETGROUP
CREATE FILE filename /GROUP command B-2
VTF
CREATE FILE filename /VTF command B-2

commands, specific
CREATE FILE B-1
RELOCATE A-6
SHOW SYSTEM A-6

commenting key commands B-21

concatenating keys B-21

connect token 4-2

cr_buf argument 5-50

CREATE FILE command
- /ASCII parameter modifier B-3
- /ATF parameter modifier B-4
- /CACHE parameter modifier B-4
- /DIRECT parameter modifier B-5
- /EDC parameter modifier B-5
- /EXTERNAL parameter modifier B-5
- /FSET parameter modifier B-6
- /GROUP parameter modifier B-6
- /LIMIT parameter modifier B-7
- /NEWPASSWORDS parameter modifier B-7
- /PASSWORDS parameter modifier B-7
- /REPLACE parameter modifier B-8
- /REPORT command modifier B-3
- /RETENTION parameter modifier B-9
- /SIZE parameter modifier B-9
- /TYPE parameter modifier B-10
- /VSET parameter modifier B-11
- /VTF parameter modifier B-11
- description B-1
- filename parameter B-3

c-token argument
- LSMCK general usage function 5-55
- LSMCO file operation function 5-15
- LSMCON session control function 5-3
- LSMDIS session control function 5-5
- LSMOS file operation function 5-8
- LSMOV file operation function 5-21
- LSMSCI StorHouse command submission function 5-50

current record positions
- key 3-2
- sequential 3-2

D

data extent size, estimating B-12

data table size B-13

data transfer control functions
- LSMCH 5-47
- LSMDEL 5-46
- LSMR 5-31
- LSMRK 5-37
- LSMRNK 5-39
- LSMRR 5-35
- LSMRS 5-33
- LSMSCI 5-50
- LSMW 5-41
- LSMWK 5-43

data transfer functions 1-3

data transfer link identifier (o_token) 1-3, 2-1

DD= characters in strings 4-2, 5-8

default access group for account 2-2

default access rights for account 2-2

defining keys B-20

definitions
- access privilege 2-2
- account (StorHouse) 2-1
- account identification code (AID) 2-1
- account password (StorHouse) 2-2
- command privilege 2-2
- external keys B-19
- file (StorHouse) 2-3
- file access group (StorHouse) 2-4
- file name (StorHouse) 2-3
- file revision (StorHouse) 2-6
- file version (StorHouse) 2-5
- internal keys B-19

DELETE command privilege
- CREATE FILE filename /LIMIT command B-2
- CREATE FILE filename /REPLACE command B-2

DF extent size, estimating B-13

directory information for files 2-6

E

entry sequence for records 3-1

estimating /SIZE value
- data extent size B-12
- DF extent size B-13
- K extent size B-14
- total B-12

examples
- checkpoint/restart A-1

programming A-5
record sequencing 3-2

external key B-19

externally specified parameters 4-2

F

file (StorHouse) 2-3

file access group (StorHouse) 2-4

file access passwords 2-4

file directory information 2-6

file name (StorHouse) 2-3

file operation functions
LSMCLO 1-3, 5-28
LSMCO 1-2, 5-14
LSMCP 1-3, 5-26
LSMOS 1-2, 5-7
LSMOV 1-2, 5-20

file positioning
entry sequence 3-1
key sequence 3-1

file revision (StorHouse) 2-6

file size, estimating B-12

file version (StorHouse) 2-5

file_attrib argument
LSMCO file operation function 5-16
LSMOS file operation function 5-9
LSMOV file operation function 5-22

file_location argument
LSMCO file operation function 5-16
LSMOS file operation function 5-9

file_name argument
LSMCO file operation function 5-15
LSMOS file operation function 5-8
LSMOV file operation function 5-21

file_options argument 5-11

file_password argument 5-15

file_passwords argument
LSMOS file operation function 5-9
LSMOV file operation function 5-21

filename parameter
CREATE FILE command B-3

files with keys B-13

FILETEK.INI File C-3

FORTRAN language 5-10

functions
categories 5-2
data transfer control 5-30
file operation 5-6
session control 5-2
StorHouse command submission 5-49

G

general usage functions
LSMAB 5-61
LSMASY 5-57
LSMCK 5-55
LSMMSG 5-59

group access passwords 2-4

group_name argument
LSMCO file operation function 5-15
LSMOS file operation function 5-9
LSMOV file operation function 5-21

group_password argument 5-15

group_passwords argument
LSMOS file operation function 5-9
LSMOV file operation function 5-22

guidelines
checkpoint/restart A-1
programming A-4
user A-6

I

index area size B-14

indicative text messages 4-3

installing StorHouse APIs
OS/2 D-1
Windows C-1

internal key B-19

K

K extent size, estimating B-14

key argument 5-43

key commands, commenting B-21

key data area size B-14

key definition components B-20

key index table B-14

key index table size B-14

key name table B-13

key segment location table size B-13

key sequence for records 3-1

key_length argument
 LSMRK data transfer control function 5-37
 LSMWK data transfer control function 5-43

key_name argument 5-37

key_value argument 5-37

key-definition mode B-19

keys
 concatenating B-21
 defining B-20

keysequential files B-14

L

link identifiers
 data transfer 2-1
 session 2-1

LSMAB general usage function
 any-token argument 5-61
 description 5-61
 overview 5-61
 return codes 5-61

LSMASY general usage function
 any-token argument 5-57
 description 5-2, 5-57
 overview 5-57
 p1_value argument 5-57
 return codes 5-57
 set_flag argument 5-57

LSMCH data transfer control function
 description 5-47
 o-token argument 5-47
 overview 5-47
 return codes 5-47

LSMCK general usage function
 any-token argument 5-55
 description 5-55
 overview 5-55
 return codes 5-55

LSMCLO file operation function
 description 5-29
 o-token argument 5-28
 overview 1-3, 5-28
 return codes 5-28

LSMCO file operation function
 c-token argument 5-15
 description 5-19
 file_attrib argument 5-16
 file_location argument 5-16
 file_name argument 5-15
 file_password argument 5-15
 group_name argument 5-15
 group_password argument 5-15
 message_flag argument 5-15
 model_file_name argument 5-16
 o_token argument 5-15
 overview 1-2, 5-14
 return codes 5-18

LSMCON session control function
 account argument 5-3
 c-token argument 5-3
 description 5-4
 message_flag argument 5-3
 overview 1-2, 5-3
 password argument 5-3
 return codes 5-4
 sm_identifier argument 5-4
 subsystem_identifier argument 5-4

LSMCP file operation function
 description 5-26
 o-token argument 5-26
 overview 1-3, 5-26
 return codes 5-26
 return_ckpt_num argument 5-26

LSMDEFS include file 4-1

LSMDEL data transfer control function
- description 5-46
- o-token argument 5-46
- overview 5-46
- return codes 5-46

LSMDIS session control function
- c-token argument 5-5
- description 5-5
- overview 1-2, 5-5
- return codes 5-5

LSMMSG general usage function
- any-token argument 5-59
- description 5-59
- message_buffer argument 5-59
- message_buffer_size argument 5-59
- overview 5-59
- return codes 5-59
- returned_message_len argument 5-59

LSMOS file operation function
- c-token argument 5-8
- description 5-12
- file_attrib argument 5-9
- file_location argument 5-9
- file_name argument 5-9
- file_options argument 5-11
- file_passwords argument 5-9
- group_name argument 5-9
- group_passwords argument 5-9
- message_flag argument 5-8
- mode argument 5-8
- o_token argument 5-8
- overview 1-2, 5-7
- return codes 5-12
- version argument 5-9

LSMOV file operation function
- access_method argument 5-21
- c-token argument 5-21
- description 5-23
- file_attrib argument 5-22
- file_name argument 5-21
- group_name argument 5-21
- group_passwords argument 5-22
- message_flag argument 5-21
- mode argument 5-21
- o_token argument 5-21
- overview 1-2, 5-20
- rel_rec_num argument 5-22
- return codes 5-23
- revision argument 5-21

LSMR data transfer control function
- buffer argument 5-31
- buffer_size argument 5-31
- description 5-31
- o-token argument 5-31
- overview 5-31
- return codes 5-31
- return_rec_len argument 5-31

LSMRK data transfer control function
- buffer argument 5-37
- buffer_size argument 5-37
- description 5-38
- key_length argument 5-37
- key_name argument 5-37
- key_value argument 5-37
- o-token argument 5-37
- overview 5-37
- return codes 5-38
- return_rec_len argument 5-37
- return_rec_num argument 5-37
- using to read a VRAM file 3-2

LSMRNK data transfer control function
- buffer argument 5-39
- buffer_size argument 5-39
- description 5-40
- o-token argument 5-39
- overview 5-39
- return codes 5-39
- return_rec_len argument 5-39
- return_rec_num argument 5-39
- using to read a VRAM file 3-2

LSMRR data transfer control function
- buffer argument 5-35
- buffer_size argument 5-35
- description 5-36
- o-token argument 5-35
- overview 5-35
- return codes 5-35
- return_rec_len argument 5-35
- return_rec_num argument 5-35
- using to read a VRAM file 3-2

LSMRS data transfer control function
- buffer argument 5-33
- buffer_size argument 5-33

description 5-34
o-token argument 5-33
overview 5-33
return codes 5-33
return_rec_len argument 5-33
return_rec_num argument 5-33
using to read a VRAM file 3-2

LSMS_TOKEN structure 4-2

LSMSCI StorHouse command interface function
cr_buf argument 5-50
c-token argument 5-50
description 5-52
overview 5-50
resp_buf argument 5-51
resp_bufsize argument 5-51
resp_info argument 5-51
return codes 5-51

LSMW data transfer control function
buffer argument 5-41
description 5-41
o-token argument 5-41
overview 5-41
record_length argument 5-41
return codes 5-41
return_rec_num argument 5-41

LSMWK data transfer control function
buffer argument 5-43
description 5-44
key argument 5-43
key_length argument 5-43
o-token argument 5-43
overview 5-43
record_length argument 5-43
return codes 5-43
return_rec_num argument 5-43

LSMxxx
function categories 5-2
functions 1-1, 5-1
sample program 6-1

M

message_buffer argument 5-59

message_buffer_size argument 5-59

message_flag argument
LSMCO file operation function 5-15
LSMCON session control function 5-3
LSMOS file operation function 5-8
LSMOV file operation function 5-21

Microsoft C 6.0 compiler and link switches D-3

mode argument
LSMOS file operation function 5-8
LSMOV file operation function 5-21

model_file_name argument 5-16

modes of operation
asynchronous 5-1
synchronous 5-1

N

next-key read operations 3-1

O

o_token argument
LSMCO file operation function 5-15
LSMOS file operation function 5-8
LSMOV file operation function 5-21

o_token data transfer link identifier 1-3, 2-1

open token 4-2

OS/2 StorHouse API
compiling and linking with API D-2
description D-1
installation disk contents D-1
installation requirements D-1
installation steps D-2
linking D-3
writing a program using API D-3

o-token argument
LSMCH data transfer control function 5-47
LSMCLO file operation function 5-28
LSMCP file operation function 5-26
LSMDEL data transfer control function 5-46
LSMR data transfer control function 5-31
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33

LSMW data transfer control function 5-41
LSMWK data transfer control function 5-43

P

p1_value argument 5-57

parameter modifiers
/ASCII
CREATE FILE command B-3
/ATF
CREATE FILE command B-4
/CACHE B-4
/DIRECT B-5
/EDC
CREATE FILE command B-5
/EXTERNAL B-5
/FSET
CREATE FILE command B-6
/GROUP
CREATE FILE command B-6
/LIMIT
CREATE FILE command B-7
/NEWPASSWORDS
CREATE FILE command B-7
/PASSWORDS
CREATE FILE command B-7
/REPLACE
CREATE FILE command B-8
/RETENTION
CREATE FILE command B-9
/SIZE
CREATE FILE command B-9
/TYPE B-10
/VSET
CREATE FILE command B-11
/VTF
CREATE FILE command B-11

parameters
filename
CREATE FILE command B-3

password argument 5-3

PASSWORD command privilege
CREATE FILE filename /NEWPASSWORDS command B-2

passwords
account 2-2
file access 2-4
group access 2-4

printable ASCII characters 2-3

privileges (StorHouse)
access 2-2
command 2-2

program fixes, guidelines A-6

programming examples A-5

programming guidelines A-4

R

read functions for VRAM files
LSMRK 3-2
LSMRNK 3-2
LSMRR 3-2
LSMRS 3-2

RECORD command privilege
CREATE FILE command B-1

record modification table size B-14

record sequencing
entry 3-1
key 3-1

record_length argument
LSMW data transfer control function 5-41
LSMWK data transfer control function 5-43

rel_rec_num argument 5-22

RELOCATE command A-6

requirements for applications to perform StorHouse operations 1-2

resp_buf argument 5-51

resp_bufsize argument 5-51

resp_info argument 5-51

return codes 4-2
LSMAB general usage function 5-61
LSMASY general usage function 5-57
LSMCH data transfer control function 5-47
LSMCK general usage function 5-55
LSMCLO file operation function 5-28
LSMCO file operation function 5-18

LSMCON session control function 5-4
LSMCP file operation function 5-26
LSMDEL data transfer control function 5-46
LSMDIS session control function 5-5
LSMMSG general usage function 5-59
LSMOS file operation function 5-12
LSMOV file operation function 5-23
LSMR data transfer control function 5-31
LSMRK data transfer control function 5-38
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33
LSMSCI StorHouse command interface function 5-51
LSMW data transfer control function 5-41
LSMWK data transfer control function 5-43

return_ckpt_num argument 5-26

return_rec_len argument
LSMR data transfer control function 5-31
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33

return_rec_num argument
LSMRK data transfer control function 5-37
LSMRNK data transfer control function 5-39
LSMRR data transfer control function 5-35
LSMRS data transfer control function 5-33
LSMW data transfer control function 5-41
LSMWK data transfer control function 5-43

returned_message_len argument 5-59

revision argument 5-21

roll back for file revisions A-3

S

sample program 6-1

session control functions
LSMCON 1-2, 5-3
LSMDIS 1-2, 5-5

session identifier (c_token) 1-2, 2-1

session identifiers 2-1

set_flag argument 5-57

SETGROUP command privilege
CREATE FILE filename /GROUP command B-2

SHOW SYSTEM command A-6

size
change table B-14
checkpoint table B-13
data extent B-12
data table B-13
DF extent B-13
index area B-14
key data area B-14
key index table B-14
key name table B-13
key segment location table B-13
record modification table B-14

sm_identifier argument 5-4

StorHouse
access privileges for account 2-2
account 2-1
account identification code (AID) 2-1
account password 2-2
API for OS/2 D-1
API for Windows C-1
command functions 1-3
command privileges for account 2-2
CREATE FILE command B-1
default access group 2-2
default access rights 2-2
file 2-3
file access group 2-4
file name 2-3
file revision 2-6
file version 2-5

subsystem_identifier argument 5-4

synchronous mode 5-1

T

tokens
connect 4-2
open 4-2

transportable files 2-6

U

updated files B-14

user guidelines A-6

V

version argument 5-9

VTF command privilege
    CREATE FILE filename /VTF command B-2

W

Windows StorHouse API
    compiling and linking the sample code C-2
    description C-1
    installation disk contents C-2
    installation requirements C-1