NetBackup™ DataStore SDK Programmer's Guide for XBSA 1.1.0

Release 10.0

VERITAS™

NetBackup™ DataStore SDK Programmer's Guide for XBSA 1.1.0

Last updated: 2022-02-25

Legal Notice

Veritas Technologies LLC
2625 Augustine Drive
Santa Clara, CA 95054

http://www.veritas.com

Technical Support

Technical Support maintains support centers globally. All support services will be delivered in accordance with your support agreement and the then-current enterprise technical support policies. For information about our support offerings and how to contact Technical Support, visit our website:

https://www.veritas.com/support

You can manage your Veritas account information at the following URL:

https://my.veritas.com

If you have questions regarding an existing support agreement, please email the support agreement administration team for your region as follows:

Worldwide (except Japan)	CustomerCare@veritas.com
Japan	CustomerCare_Japan@veritas.com

Documentation

Make sure that you have the current version of the documentation. Each document displays the date of the last update on page 2. The latest documentation is available on the Veritas website:

https://sort.veritas.com/documents

Documentation feedback

Your feedback is important to us. Suggest improvements or report errors or omissions to the documentation. Include the document title, document version, chapter title, and section title of the text on which you are reporting. Send feedback to:

NB.docs@veritas.com

You can also see documentation information or ask a question on the Veritas community site:

http://www.veritas.com/community/

Veritas Services and Operations Readiness Tools (SORT)

Veritas Services and Operations Readiness Tools (SORT) is a website that provides information and tools to automate and simplify certain time-consuming administrative tasks. Depending on the product, SORT helps you prepare for installations and upgrades, identify risks in your datacenters, and improve operational efficiency. To see what services and tools SORT provides for your product, see the data sheet:

https://sort.veritas.com/data/support/SORT_Data_Sheet.pdf

Contents

Chapter 1

Introduction to NetBackup XBSA

This chapter includes the following topics:

About Introduction to NetBackup XBSA

The applications or the facilities that need data storage management for backup or archive purposes can use the NetBackup XBSA application programming interface (API). The XBSA API creates a backup or an archive application that communicates with NetBackup.

What is NetBackup XBSA?

XBSA is an Open Group Technical Standard that defines a Backup Services API (XBSA). The XBSA specification consists of source procedure calls, type definitions, data structures, and return codes. The client applications use these to be able to use a backup service, NetBackup, and to store and manage data.

The NetBackup XBSA is an API to NetBackup developed to the XBSA specifications. The NetBackup XBSA interface has extended the XBSA specifications to make it

easier to use and enhance performance when used with NetBackup. Exceptions are noted throughout the document.

See "Resources" on page 11.

NetBackup XBSA is provided as a Software Developers Kit (SDK) that includes the header files and the libraries that are required to create an XBSA application.

What does NetBackup XBSA do?

The NetBackup XBSA interface allows an XBSA application to create, query, retrieve, and delete data objects using NetBackup for data storage. The operations on the objects use the rules and the policies that NetBackup defines and enforces. Examples of these rules and policies include the type of media the objects are stored on, the number of copies, the retention policies, the scheduled operations, and so on.

Objects are created and retrieved as a stream of data. Each object also has a set of attributes that are used to describe the object. These attributes include a CopyId, created by the NetBackup XBSA interface, that uniquely defines the object. The XBSA application specifies and uses other attributes to describe the object. When an object is retrieved, it is returned as a data stream and the XBSA application restores it to its original form.

An XBSA application can also query the NetBackup XBSA interface for the objects that it owns. This query is based on a subset of the attributes that were specified. The result of a query is a list of objects and their attributes. It can also be an empty list.

Objects can also be deleted when the XBSA application no longer needs them. Deleting an object prevents it from being retrieved or queried but does not necessarily delete the data. When the actual data gets deleted is a function of NetBackup.

Terminology

The fundamental terms necessary to understand this NetBackup XBSA are described in the following table.

Table 1-1 XBSA Terms

Term	Definition
XBSA application	Application-specific software that uses the NetBackup XBSA API to request NetBackup services. Typically, an XBSA application is tightly bound to a user application (such as a DBMS) or an operating system service (such as a file system).
NetBackup XBSA interface	The NetBackup software that communicates with NetBackup to carry out the functions that are defined by this specification.
NetBackup XBSA environment	The NetBackup XBSA environment is the environment that exists between the NetBackup XBSA interface and the XBSA application. A NetBackup XBSA session defines this environment. NetBackup XBSA environment variables are used to pass specific NetBackup information between the XBSA application and the NetBackup XBSA interface. Setting platform environment variables (such as `getenv` or `setenv`) has no effect on the NetBackup XBSA environment.
NetBackup XBSA session	A NetBackup XBSA session is a logical connection between an XBSA application and NetBackup XBSA interface. A session begins with a call to BSAInit() and ends with a call to BSATerminate(). Nested sessions are not supported.
NetBackup XBSA object	The NetBackup XBSA API uses an object-based paradigm. Every data object visible and transferred at the NetBackup XBSA interface is a NetBackup XBSA object. The XBSA application defines the objects that it backs up and restores.

Important concepts

To get the most out of using the NetBackup XBSA interface, a working knowledge of NetBackup is required. When the XBSA application controls NetBackup concepts, such as policy, schedule, time-outs, and multiplexing, it can be more robust and perform better in a NetBackup environment. Other items, like storage units, determine where the data gets stored and that can affect the XBSA application.

Note: The NetBackup XBSA interface does not provide an interface for managing the configuration, media, jobs, and so on. These types of operations must be done through other NetBackup command line or graphical users interface.

Resources

The NetBackup XBSA API specification is based on the Open Group Technical Standard for Systems Management: Backup Services API (XBSA) Document Number: C425. More information on this standard can be found at the following URL:

http://archive.opengroup.org/publications/archive/CDROM/c425.pdf

Chapter 2

How to set up the SDK

This chapter includes the following topics:

System requirements

The following items are needed before setting up the SDK.

- Supported systems. See the Database Agent Compatibility list for NetBackup Enterprise Server for a list of platforms that are supported with XBSA. This list is available at the following URL:
  http://www.netbackup.com/compatibility
- ANSI-compatible compiler
- To develop an application, you need NetBackup, a DataStore License Key, and the NetBackup DataStore SDK installed.
- To run an application, you need a NetBackup client installed (on the client running the XBSA application) and the DataStore License Key (on the NetBackup server).

Installing the SDK

The NetBackup for DataStore SDK is released on a separate CD from the rest of NetBackup. You must have this CD to install the SDK. When the SDK is installed, the files should be moved to the environment where the development of the XBSA application is to be done.

Installation requirements

The following items are required before you install the SDK:

- NetBackup 10.0 server software is installed and operational on the server where the SDK is to be installed.
- Adequate disk space (approximately 20 M) on the server must be present to receive the software.

Installation instructions for UNIX platforms

To install the SDK on UNIX platforms

1. Log on as the root user on the computer.

   If you are already logged on, but are not the root user, execute the following command.

   ```
   su - root
   ```

2. Verify that a registered and valid license key resides on the master server.

   To view or add license keys, perform one of the following:

   - Run the following command:

     ```
     /usr/openv/netbackup/bin/admincmd/get_license_key
     ```

     If you run the `get_license_key` command from the master server, it returns the correct information by default.
     If you run the `get_license_key` command from the media server, specify the master server's host name as the computer you are querying. If you do not specify the master server's host name, the command returns licensing information about the media server.

   - Open the NetBackup Administration Console and choose **Help > License Keys**.

3. Insert the NetBackup DataStore SDK CD-ROM into the drive.

4 Change the working directory to the CD-ROM directory.

```
cd /CD_mount_point
```

5 Load and install the software by executing the install script.

```
./install
```

A prompt appears asking if the package is correct.

Answer `y`.

The SDK files are extracted into the directory *install_path*`/openv/netbackup/sdk`.

The file `version_dstore` is extracted into the directory *install_path*`/openv/share`.

Installation instructions for Windows platforms

To install the SDK on Windows platforms

1 Insert the CD-ROM into the drive.

- On systems with Autoplay enabled for CD-ROM drives, the install program starts automatically.
- On the systems that have Autoplay disabled, click the **Start** option and choose **Run**. Type `D:\Autorun\AutoRunI.exe`, where D:\ is your CD-ROM drive.

2 Follow the prompts throughout the wizard.

Uninstalling the SDK

The NetBackup for DataStore SDK is delivered in native packaging format. Remove the NetBackup for DataStore SDK by executing the native command appropriate for your operating system:

- AIX: `installp -u VRTSnbsds`
- HP-UX: `swremove VRTSnbsds`
- Linux: `rpm -e VRTSnbsds`
- Solaris: `pkgrm VRTSnbsds`

Windows: On the **Control Panel**, select **Programs and Features**. Select **Veritas NetBackup - DataStore**, then click **Remove**.

Configuration

Creating an XBSA application using the NetBackup XBSA SDK should require a minimum of setup. The SDK is installed as read only in the NetBackup directory. The files that are used should be moved to the development environment of the application.

The sample directory provides a Makefile for UNIX platforms and one for Windows platforms. They create valid executables for the sample programs, but they should only be used as guides. The developers should use the compile options and libraries that are optimal for their application. The XBSA libraries and header files themselves do not require any special options.

Description of the XBSA SDK package

The NetBackup SDK contains the libraries with the XBSA interfaces for each of the platforms that the SDK supports. Header files are required to compile an XBSA application. The SDK is installed in the NetBackup directory, either `/usr/openv/netbackup/sdk/DataStore/XBSA` on UNIX or *`install_directory`*`\VERITAS\NetBackup\sdk\DataStore\XBSA` on Windows. This directory contains all of the files necessary to build an XBSA application.

The package contains the following directories.

Table 2-1 SDK/DataStore/XBSA Directories

Directory	Description
samples	Contains sample programs and scripts.
lib	Contains the library files for each supported system.
include	Contains the header files.

Library files

The NetBackup XBSA SDK contains the archive libraries for each of the systems. Installed with the NetBackup client is an XBSA shared object library. This allows the developer to choose the method of binding for each application. Both of these libraries contain all XBSA functions and all external references.

The XBSA libraries are found in the directory `/usr/openv/netbackup/sdk/DataStore/XBSA/lib`. In this directory is a directory for each hardware type. Within each of these directories is a directory for each supported operating system level. For UNIX operating systems, there is the

`libxbsa.a` library. For the Windows operating systems, there is both an `xbsa.lib` and a `xbsas.lib`. The `xbsa.lib` was generated when creating the `xbsa.dll` and `xbsas.lib` is a full static library.

Header files

Two header files are released with the SDK. These should be used when you compile the XBSA application. These header files are found in the `/usr/openv/netbackup/sdk/DataStore/XBSA/include` directory.

Table 2-2 Header Files

File	Description
`xbsa.h`	Header file that contains the XBSA defined structures.
`nbbsa.h`	Header file that contains NetBackup specific definitions for the NetBackup XBSA interface.

Chapter 3

Using the NetBackup XBSA interface

This chapter includes the following topics:

Getting help with the API

While working with the API, you can obtain reference information about the XBSA functions.

See “Error messages” on page 76.

Sample applications are included with XBSA.

See “What the sample files do” on page 136.

NetBackup XBSA data structures

This section describes the XBSA data structures and explains how the NetBackup XBSA interface and the XBSA application use them for creating and manipulating XBSA objects.

Object data

NetBackup XBSA object data contains the actual data entity that is archived or backed up by an XBSA application. The NetBackup XBSA API supports only one type of object data, which is a variable-length, unstructured and uninterpreted byte-stream.

To a specific XBSA application, however, the XBSA object data can contain an internal structure that reflects the data of the application objects that the XBSA application has archived or backed up. In this context, the XBSA object data can contain one of the following examples: a UNIX file system, a UNIX directory, a file, a document, a disk image, a data stream, or a memory dump.

Through the NetBackup XBSA interface, object data can be stored, retrieved, or deleted, but not searched or modified. Since the object data can be stored on slow (or offline) media, it is generally not advisable for an XBSA application to store metadata in object data, especially the information that can influence a data-retrieval decision.

However, the metadata of an XBSA object, that is stored in the catalog, can be replicated in its object data if it enhances the performance of the object restore. This is an XBSA application implementation decision.

Object descriptors

A NetBackup XBSA object has a BSA_ObjectDescriptor, that contains cataloging information and optional application-specific object metadata. Cataloging information is capable of interpretation and searching by the NetBackup XBSA interface. Application-specific object metadata is not interpretable by the NetBackup XBSA interface but it can be retrieved and interpreted by an application. Using an object's objectName or its assigned copyId identifier, the corresponding BSA_ObjectDescriptor and object data can be retrieved through the NetBackup XBSA interface.

A BSA_ObjectDescriptor consists of a collection of object attributes. The basic data types used for XBSA object attributes are:

- Fixed-length character strings
- Hierarchical character strings (with a specified delimiter, and a length limit on the overall string)
- Enumerations
- Integers (with a specified range limit)
- Date-time (in a standard C TM structure) format and precision; for example, *yyyymmddhhmm*)

The attributes are shown in the following table:

Table 3-1 BSA_ObjectDescriptor Attributes

Attribute	Data Type	Searchable
objectOwner	(consisting of two parts)	Yes
bsa_ObjectOwner	[fixed-length character string]	
app_ObjectOwner	[hierarchical character string]	
objectName	(consisting of two parts)	Yes
objectSpaceName	[fixed-length character string]	
pathName	[hierarchical character string]	
createTime	[date-time]	Yes
copyType	[enumeration]	Yes
copyId	64-bit unsigned integer	No
restoreOrder	64-bit unsigned integer	No
resourceType	[fixed-length character string]	No
objectType	[enumeration]	Yes
objectStatus	[enumeration]	Yes
objectDescription	[fixed-length character string]	No
estimatedSize	[64-bit unsigned integer]	No
objectInfo	[fixed-length byte string]	No

Each NetBackup XBSA object is a copy of certain application object(s):

- To preserve the semantics of the use of each copy within the BSA_ObjectDescriptor, each NetBackup XBSA object has a copyType of either backup or archive. The NetBackup XBSA interface recognizes the copyType so that the two types of objects can be managed differently and accessed separately.

  Note: It is up to the XBSA application to manage these types differently, as the NetBackup XBSA interface only keeps track of the type of the object.

- Each NetBackup XBSA object also has an objectStatus of either most_recent or not_most_recent.
- To capture an application object's type information, the corresponding NetBackup XBSA object can have a resourceType (for example, "filesystem") and a possible resource-specific BSA_ObjectType (for example, BSA_ObjectType_FILE).

An XBSA application can search for a NetBackup XBSA object within a certain search scope (for example, among objects in a certain objectSpaceName). It qualifies the search on the value of the appropriate searchable attributes.

On the other hand, non-searchable, application-specific attributes can be provided by an XBSA application for storage in the BSA_ObjectDescriptor, but the NetBackup XBSA interface does not interpret these attributes. They are stored in the NetBackup XBSA object attributes objectInfo, resourceType, and objectDescription.

The objectInfo field defaults to a character string. It can also be used to store binary data by using the NBBSA_OBJINFO_LEN XBSA environment variable.

Through these descriptor attributes, application-specific metadata can be stored in the catalog. Then, this metadata can be efficiently retrieved without retrieving the actual object data that is stored in the repository. An XBSA application can use these attributes to maintain inter-object relationships and dependencies. Some consideration should be given as to how much data is stored in the NetBackup Catalog. The amount of metadata that is stored with a few large objects can be larger than the amount that is stored for a million small objects.

Query descriptors

A BSA_QueryDescriptor is the structure that is used in the query process to find an individual or set of objects. It contains those fields from the object descriptor that are searchable. When a query is performed, the enumeration fields must be specified. If they are unknown, they all allow an "ANY" enumeration. You must also specify the objectName.pathName. Wildcards are allowed for this field and "/*" is a valid pathname for querying. The other strings in the descriptor can be empty strings, but they are still used for comparison to find an object descriptor that matches the query descriptor. If these fields are unknown, wildcards are allowed here also. The start (createTime_from) and end (createTime_to) dates are not required.

The attributes of the BSA_QueryDescriptor are shown in the following table:

Table 3-2 BSA_QueryDescriptor Attributes

Attribute	Data Type
objectOwner	(consisting of two parts)

Table 3-2 BSA_QueryDescriptor Attributes *(continued)*

Attribute	Data Type
bsa_objectOwner	[fixed-length character string]
app_objectOwner	[hierarchical character string]
objectName	(consisting of two parts)
objectSpaceName	[fixed-length character string]
pathName	[hierarchical character string]
createTime_from	[date-time]
createTime_to	[date-time]
CopyType	[enumeration]
objectType	[enumeration]
objectStatus	[enumeration]

Note: The createTime_from and createTime_to fields are not part of the XBSA specification for the BSA_QueryDescriptor structure. The NetBackup XBSA interface uses two reserved fields from the BSA_QueryDescriptor structure to allow this information to be used (if available) for the query. These fields are not required, although if the XBSA application can specify these dates, it can, in some instances, greatly speed up query time.

Buffers

The XBSA application allocates all of the buffers that the NetBackup XBSA interface uses. The NetBackup XBSA interface fills data into the buffers, but never allocates any memory that is passed back to the XBSA application. This process simplifies buffer allocation and deletion since the XBSA application is solely responsible.

The API uses several conventions that let the NetBackup XBSA interface influence how buffers are allocated and provide an interface with the ability to reserve private sections in certain buffers.

Buffer size

When API calls specify the size of the buffer as a separate parameter, the interface uses the following convention to signal that a buffer is not large enough and provides the XBSA application with a way to determine the correct size.

The parameter that specifies the size is a pointer, so that the NetBackup XBSA interface can alter the parameter. The size is always in bytes. If the size is adequate and a valid buffer is given, the NetBackup XBSA interface copies the requested data into the buffer and sets the actual size in the size parameter.

If the size is inadequate, the NetBackup XBSA interface does not copy the data into the buffer. It sets the size parameter to the actual size of the data to be copied and returns from the function call with BSA_RC_BUFFER_TOO_SMALL. This allows the XBSA application to allocate a buffer of adequate size and to call the function again.

The functions that use this convention are BSAGetEnvironment(), NBBSAGetEnv() and BSAQueryServiceProvider().

Private buffer space

When function calls use the BSA_DataBlock32 structure, a convention lets the NetBackup XBSA interface reserve certain portions of the buffer for its own use. The NetBackup XBSA interface can reserve the following areas:

Header	A contiguous area starting at offset 0 (that is, the start of the buffer)
Trailer	A contiguous area that ends at the end of the buffer (that is, the tail of the buffer)

The area that is reserved for the XBSA application is as follows:

Data Segment	A contiguous area that lies in between the Header and Trailer

To make this preference known to the XBSA application, the NetBackup XBSA interface can set certain parameters in the BSA_DataBlock32 structure when a data transfer is initiated. Specifically, when the XBSA application issues either the BSACreateObject() call or the BSAGetObject() call, the BSA_DataBlock32 structure is used for passing the preference of the NetBackup XBSA interface. The parameters set by the NetBackup XBSA interface are described in the following table:

Table 3-3 Parameters in the BSA.DATABlock32 Structure

Parameter	Preference
bufferLen == 0	The interface has no restrictions on the buffer length. No trailer portion is required.

Table 3-3 Parameters in the BSA.DATABlock32 Structure *(continued)*

Parameter	Preference
bufferLen != 0	The interface accepts the buffers that are at least bufferLen bytes in length (minimum length). It also accepts larger buffers. For a BSASendData() call, the interface accepts a trailer that is as least as large as: trailerBytes >= (bufferLen - numBytes - headerBytes) For a BSAGetData() call, the interface returns a trailer that is not larger than: trailerBytes <= (bufferLen - numBytes - headerBytes)
numBytes == 0	The interface has no restrictions on the length of the data portion of the buffer.
numBytes != 0	The interface accepts (for a BSASendData() call), or returns (for a BSAGetData() call), a data segment that does not exceed numBytes bytes.
headerBytes == 0	The interface only accepts or returns buffers with no header.
headerBytes != 0	The length of the header portion of the buffers that are accepted or returned by the interface is headerBytes bytes.
bufferPtr	Not used

Subsequent calls to BSAGetData() or BSASendData() must adhere to the preferences that are specified by the NetBackup XBSA interface.

The NetBackup XBSA interface can write anything into the header and trailer area of the actual buffer, as specified by the bufferPtr parameter in the BSA_DataBlock32 structure.

The NetBackup XBSA interface has a buffer size limit of 1 Gigabyte.

Note: For NetBackup XBSA Version 1.1.0, there are no header or trailer requirements. The XBSA specifications define the format that is documented here and can be used in the future by NetBackup.

Use of BSA_DataBlock32 in BSASendData()

For BSASendData(), all parameters in the BSA_DataBlock32 structure are set by the XBSA application and adhere to the NetBackup XBSA interface preferences or the function fails with a BSA_RC_INVALID_DATABLOCK error. The NetBackup XBSA interface is not allowed to change any of the parameters.

The buffers that are passed by BSASendData() must be full. This means that numBytes must be equal to bufferLen. The buffer for the last BSASendData() call for an object does not need to be full.

Use of BSA_DataBlock32 in BSAGetData()

For BSAGetData(), all parameters in the BSA_DataBlock32 structure must be set by the XBSA application and adhere to the NetBackup XBSA interface preferences or the function fails with a BSA_RC_INVALID_DATABLOCK error. The NetBackup XBSA interface changes the numBytes parameter setting to the actual number of bytes copied into the data segment. NetBackup is not allowed to change any of the other parameters.

Shared memory

Note: The passing of data in shared memory blocks between the XBSA application and the NetBackup XBSA interface is not supported for NetBackup XBSA Version 1.1.0.

The BSA_DataBlock32 structure contains fields to allow the use of shared memory blocks for passing data between an XBSA application and the NetBackup XBSA interface. The shareId and shareOffset fields of the BSA_DataBlock32 structure are used to define shared memory buffers. The NetBackup XBSA interface version 1.1.0 does not use these fields.

NetBackup XBSA environment

The NetBackup XBSA environment is created when an XBSA application calls BSAInit() to initiate a session. This environment only exists between the NetBackup XBSA interface and the XBSA application. The XBSA environment variables are used to pass specific NetBackup information in both directions between the XBSA application and the NetBackup XBSA interface. The environment variables are generally set or modified by the XBSA application, but the NetBackup XBSA interface creates and modifies some variables to pass information back to the XBSA application. Setting platform environment variables (getenv or setenv) has no effect on the NetBackup XBSA environment.

There are restrictions when some of the variables can be set or modified. Most of them can be set on the call to BSAInit(), which initiates a session. Some can also be modified within a session but outside of a transaction. And a few can be modified within a transaction. These limitations are outlined in the following descriptions for each of the variables.

Each XBSA environment variable is defined as a keyword that is followed by an "=" and followed by a null-terminated value. No spaces are allowed around the "=". "BSA_API_VERSION=1.1.0" is valid while "BSA_API_VERSION = 1.1.0" is not.

The functions used to create, modify, and view these environment variables are:

- BSAInit()
- BSAGetEnvironment()
- NBBSAUpdateEnv()
- NBBSASetEnv()
- NBBSAGetEnv()

See "Function specifications" on page 80.

Environment variable definitions

The following XBSA environment variables are defined as part of the XBSA specification and are accepted by the NetBackup XBSA interface.

Table 3-4 XBSA Environment Variables

Variable Name	Description	Format
BSA_API_VERSION	Mandatory. Specifies the version of the specification that the calling XBSA application requires. BSAQueryApiVersion() can retrieve the value of the current NetBackup XBSA interface.	A string containing 3 numeric elements, (version, issue, level) separated by periods.
BSA_DELIMITER	Optional. The delimiter that is used in hierarchical character strings (default "/").	A single ASCII character.
BSA_SERVICE_PROVIDER	Optional. Identifies the XBSA implementation. BSAQueryServiceProvider() can retrieve this value.	A hierarchical character string with 3 fields.
BSA_SERVICE_HOST	Optional. Identifies a specific host system for the NetBackup server.	A string containing a host name.

In addition to the environment variables that are defined in the XBSA specification, the following NetBackup XBSA environment variables are defined as part of this specification. These are specific to NetBackup and are not relevant to other XBSA implementations. See the NetBackup System Administrator's Guide, Volume I, for a more complete definition of the NetBackup policy, schedule, and logging. The NetBackup environment variables all are prefaced with "NB."

Table 3-5 NetBackup Environment Variables

Variable Name	Description	Format
NBBSA_CLIENT_HOST	Optional. Identifies a specific host system for the NetBackup client.	A string containing a host name.
NBBSA_DB_TYPE	Optional. Specifies a specific policy type.	A string containing the policy type.
NBBSA_FEATURE_ID	Optional. Specifies a specific NetBackup licensed feature within the DataStore policy type.	An integer value.
NBBSA_KEYWORD	Optional. Specifies the NetBackup Keyword field for this image.	A string containing a keyword value <= 100 characters.
NBBSA_LOG_DIRECTORY	Optional. Identifies the name of directory that contains the log files of the XBSA application.	A string containing a single directory name.
NBBSA_OBJECT_GROUP	Optional. This variable is used to define the object group owner of an object being created.	A string containing the group.
NBBSA_OBJECT_OWNER	Optional. This variable is used to define the object owner of an object being created.	A string containing the owner.
NBBSA_OBJINFO_LEN	Optional. If this variable is set before an XBSA object is created, the objectInfo field is considered to be this length and the object is considered binary.	An integer value <= 256.
NBBSA_POLICY	Optional. Identifies a specific NetBackup policy to be used.	A string containing a NetBackup policy name.
NBBSA_SCHEDULE	Optional. Identifies a specific NetBackup XBSA schedule to be used.	A string containing a NetBackup schedule name.
NBBSA_USE_OBJECT_GROUP	Optional. This variable can be set to cause the group of an object to be something other than the logon user creating the object.	An integer value between 0 and 4.
NBBSA_USE_OBJECT_OWNER	Optional. This variable can be set to cause the owner of an object to be something other than the logon user creating the object.	An integer value between 0 and 4.

The following XBSA environment variables are set by the NetBackup XBSA interface from values in the NetBackup configuration files. These environment variables are used to pass required information from NetBackup to the XBSA application. Descriptions of these NetBackup configuration values can be found in the NetBackup System Administrator's Guide, Volume I.

Table 3-6 XBSA Environment Variables for NetBackup Configuration Values

Variable Name	Description	Format
NBBSA_VERBOSE_LEVEL	The verbose level of the database logs.	An integer value between 0 and 9.
NBBSA_MULTIPLEXING	The NetBackup multiplexing value.	An integer value.
NBBSA_SERVER_BUFFSIZE	The NetBackup server buffer size value.	An integer value in bytes.
NBBSA_MEDIA_MOUNT_TIMEOUT	The NetBackup MEDIA_MOUNT_TIMEOUT value.	An integer value in seconds.
NBBSA_CLIENT_READ_TIMEOUT	The NetBackup CLIENT_READ_TIMEOUT value. This value can be modified by the XBSA application.	An integer value in seconds.

Extended environment variable definitions

Table 3-7 Extended Environment Variables

Variable Name	Extended Description
BSA_API_VERSION	BSA_API_VERSION specifies the version of the XBSA specification. The XBSA application sets it as the version that the XBSA application requires. This value is required to be in the environmental variable list in the call to BSAInit(), where it is verified as a supported version of the NetBackup XBSA interface. The current value of BSA_API_VERSION that is supported by the NetBackup XBSA interface can be retrieved with a call to BSAQueryApiVersion(). Once BSA_API_VERSION has been set in the XBSA environment, it cannot be changed by calls to NBBSAUpdateEnv() or NBBSASetEnv(). The version that is supported for this feature pack is "1.1.0."

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
BSA_DELIMITER	BSA_DELIMITER is the delimiter used in hierarchical character strings. The NetBackup XBSA interface sets this XBSA environment variable. This feature pack uses the "/" delimiter. This value can be retrieved by BSAQueryServiceProvider().
BSA_SERVICE_HOST	BSA_SERVICE_HOST identifies the host system for the NetBackup server. If this variable is not provided, the currently configured server for the NetBackup client is used. See the NetBackup System Administrator's Guide, Volume I, for information on how to use the `bp.conf` configuration file to specify the NetBackup servers. This XBSA environment variable can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.
BSA_SERVICE_PROVIDER	BSA_SERVICE_PROVIDER identifies the XBSA implementation. The NetBackup XBSA interface sets this XBSA environment variable. It is defined as: Veritas/NetBackup/1.1.0. BSAQueryServiceProvider() can also retrieve this value.
NBBSA_CLIENT_HOST	NBBSA_CLIENT_HOST identifies a specific host system as the NetBackup client. If this variable is not provided, the host on which the XBSA application is running is the client. This variable is useful for queries and restores when restoring the data that was backed up from a host other than the host where the data was restored. For backups, if the NBBSA_CLIENT_HOST is logically different from the client host where the backup is being initiated from, this results in an error, as you cannot create objects from another host. This XBSA environment variable can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_CLIENT_READ_TIMEOUT	NBBSA_CLIENT_READ_TIMEOUT is used to determine or reset the NetBackup CLIENT_READ_TIMEOUT value. The NetBackup XBSA interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). After BSACreateObject(), the NBBSA_CLIENT_READ_TIMEOUT value can be reset by the XBSA application by NBBSAUpdateEnv() or NBBSASetEnv(). Setting it at any other time has no effect. See the NetBackup System Administrator's Guide for UNIX, Volume I, or NetBackup System Administrator's Guide for Windows, Volume I, for more information about CLIENT_READ_TIMEOUT.
NBBSA_DB_TYPE	NBBSA_DB_TYPE is an internal string representation of a NetBackup policy type. This is generally only used for NetBackup internal agents, but in certain instances it can be set up for external use. If this variable is not specified, it defaults to the SDK default of DataStore policy type. If this variable is used, the NBBSA_FEATURE_ID must also be specified.
NBBSA_FEATURE_ID	NBBSA_FEATURE_ID identifies a specific NetBackup licensed feature to be used for the session. If this variable is not provided, the default DataStore feature ID is used. In general, this environment variable does not need to be set, but it allows an application, working with NetBackup product management, to use a specific NetBackup license. This value can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.
NBBSA_KEYWORD	NBBSA_KEYWORD allows the XBSA application to specify a NetBackup keyword. This keyword is typically used to group images together and can speed up a search. If this variable is specified for a backup transaction, the keyword is stored with the image. If it is specified before a query or restore transaction, the keyword is used to help in the search process. This value can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_LOG_DIRECTORY	NBBSA_LOG_DIRECTORY identifies the name of the directory that contains the log files of the NetBackup XBSA interface and possibly for the XBSA application. This directory is located in `/usr/openv/netbackup/logs` on UNIX and *`install_directory`*`\VERITAS\NetBackup\Logs` on Windows. If it is not specified, the directory name is `exten_client`. All debug messages from the NetBackup XBSA interface and from function NBBSALogMsg() go to a dated log file in this directory. This value can be set by the XBSA application by BSAInit(). It may not be modified after the call to BSAInit().
NBBSA_MEDIA_MOUNT_TIMEOUT	NBBSA_MEDIA_MOUNT_TIMEOUT is used to determine the NetBackup MEDIA_MOUNT_TIMEOUT value. The NetBackup XBSA interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). NBBSA_MEDIA_MOUNT_TIMEOUT cannot be modified by the XBSA application. See the NetBackup System Administrator's Guide, Volume I, for more information about MEDIA_MOUNT_TIMEOUT.
NBBSA_MULTIPLEXING	NBBSA_MULTIPLEXING the number of streams that NetBackup has been configured to accept at one time. The NetBackup XBSA interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). The XBSA application cannot modify NBBSA_MULTIPLEXING. See the NetBackup System Administrator's Guide, Volume I, for more information about multiplexing.
NBBSA_OBJECT_GROUP	NBBSA_OBJECT_GROUP can be used with variable NBBSA_USE_OBJECT_GROUP to define the group ownership of an object. When NBBSA_USE_OBJECT_GROUP = VxENV_OWNER, the name that is defined in this string becomes the group owner of an object that is created. This group should be a valid group name on the client. This value can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv(). It can be modified within a transaction and each object that is created within one transaction can have a different group.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_OBJECT_OWNER	NBBSA_OBJECT_OWNER can be used with variable NBBSA_USE_OBJECT_OWNER to define the ownership of an object. When NBBSA_USE_OBJECT_OWNER = VxENV_OWNER, the name that is defined in this string becomes the owner of an object that is created. This owner should be a valid user name on the client. This value can be set by the VxBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv(). It can be modified within a transaction and each object that is created within one transaction can have a different owner.
NBBSA_OBJINFO_LEN	NBBSA_OBJINFO_LEN is used by BSACreateObject() to allow the objectInfo field of the object descriptor to contain non-ASCII values. If this variable is not specified, the objectInfo field is treated as a NULL terminated character string. You do not have to specify this variable for a query or restore transaction. The XBSA application can modify this value at any time during a backup transaction using BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv(). If the length of the objectInfo field is different for each object, it can be changed before each BSACreateObject() call.
NBBSA_POLICY	NBBSA_POLICY identifies a specific NetBackup policy to be used for the transaction. If this variable is not provided, the NetBackup configuration is used to find the default policy. For backups, if a policy is configured in NetBackup on the client, that policy is used for the backup. For queries, restores, and deletes, the configured policy is not used. See the NetBackup System Administrator's Guide, Volume I, for information on how to create and configure a NetBackup policy. This value can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.
NBBSA_SCHEDULE	NBBSA_SCHEDULE identifies a specific NetBackup schedule to be used. If this variable is not provided, the NetBackup configuration is used to find the default schedule to use. For backups, if a schedule is configured in NetBackup on the client, that schedule is used for the backup. For queries, restores, and deletes, the configured schedule is not used. See the NetBackup System Administrator's Guide, Volume I, for information on how to create and configure a NetBackup schedule. This value can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_SERVER_BUFFSIZE	NBBSA_SERVER_BUFFSIZE the NetBackup configured size of the NET_BUFFER_SZ. This variable can be used by the XBSA application to help improve performance. The NetBackup XBSA interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). The XBSA application cannot modify NBBSA_SERVER_BUFFSIZE. See the NetBackup System Administrator's Guide, Volume I, for more information about setting the buffer size.
NBBSA_USE_OBJECT_GROUP	NBBSA_USE_OBJECT_GROUP lets the agent define the group owner of objects that are created with VxBSACreateObject(). The default group of an object is the logon user of the process creating the object (not the primary group of the logon user, but the actual logon user). This variable allows the agent to specify the ownership as follows. VxLOGIN_USER 0 - Default, group field is set to the logon user VxLOGIN_GROUP 1 - Group field is set to the primary group of the logon user VxBSA_OWNER 2 - Group field is set to objectDescriptor->objectOwner.bsa_ObjectOwner VxAPP_OWNER 3 - Group field is set to objectDescriptor->objectOwner.app_ObjectOwner VxENV_OWNER 4 - Group field is set to value of NBBSA_GROUP_OWNER variable This value may be set by the BSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_USE_OBJECT_OWNER	NBBSA_USE_OBJECT_OWNER allows the agent to define the owner of objects that are created with BSACreateObject(). The default ownership of an object is the logon user of the process creating the object. This variable allows the agent to specify the ownership as: VxLOGIN_USER 0 - Default, owner field is set to the logon user VxBSA_OWNER 2 - Owner field is set to objectDescriptor->objectOwner.bsa_ObjectOwner VxAPP_OWNER 3 - Owner field is set to objectDescriptor->objectOwner.app_ObjectOwner VxENV_OWNER 4 - Owner field is set to value of NBBSA_OBJECT_OWNER variable This value can be set by the XBSA application by BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but cannot be set or modified after a transaction has begun.
NBBSA_VERBOSE_LEVEL	NBBSA_VERBOSE_LEVEL is the verbose level of the NetBackup debug logs. The verbose level can be configured through the Backup, Archive, and Restore interface or the NetBackup Administration Console. This value can be useful if the XBSA application, using NBBSALogMsg(), wants to log different levels of messages to the NetBackup XBSA logs based on the verbose level that is configured in NetBackup. The NetBackup XBSA interface originally sets this value in BSAInit(). The XBSA application can reset this environment variable, using NBBSASetEnv() or NBBSAUpdateEnv(), if it wants to change the level of logging.

XBSA sessions and transactions

All operations for NetBackup must be in an XBSA session. Each session can contain one or more transactions. This section defines how the XBSA sessions are defined and what is allowed in each transaction.

Sessions

To use most of the NetBackup XBSA API calls, it is necessary for an XBSA application to set up a session with the NetBackup XBSA interface by invoking the BSAInit() call. The functions BSAQueryApiVersion() and BSAQueryServiceProvider() can be invoked before calling BSAInit(). These functions are used to determine the current version of the API used by the NetBackup XBSA interface and a string

describing the provider of the NetBackup XBSA interface, respectively, and are not dependent on being within a session.

Initialization and termination

A BSAInit() call initiates a session. This call sets up a session with the NetBackup XBSA interface and creates a context, defined by handle, for the caller to be used in subsequent calls. The XBSA environment is set up within that context and remains in place until the session is terminated. Nested sessions are not permitted.

A BSATerminate() call terminates a session, which releases any resources that are acquired during the NetBackup XBSA session. If BSATerminate() is called within a transaction, the transaction is aborted.

Authentication

In NetBackup XBSA Version 1.1.0, all authentication and security is handled by NetBackup based on the logon user. The logon user of the session that created the object determines the object ownership. To query or restore an object, the logon user doing the request must be the same user who created the object or a root administrator.

Note: NetBackup XBSA Version 1.1.0 does not validate the objectOwner and SecurityToken parameters of BSAInit(). The objectOwner fields, bsa_objectOwner and app_objectOwner, can be specified and are stored with an object, but the logon user who created the object determines the official ownership of an object. This user, or a root admin, are the only users who can query or restore this object.

Transactions

Within each session, an XBSA application can make a sequence of calls (for example, to backup some objects, to query the set of objects it has backed up, or to restore objects). These calls must be grouped into a transaction by invoking BSABeginTxn() at the beginning of the group of calls and invoking BSAEndTxn() at the end. The latter either commits the transaction or aborts it.

If a transaction is aborted either by a BSAEndTxn() or BSATerminate() call, then the effect of all of the calls that are made within the transaction is nullified. If a transaction is committed, then the effect of all the calls within the transaction is made permanent.

Within a single session, transactions cannot be nested and cannot overlap. Transactions are categorized into the following types:

- NetBackup XBSA object modification transactions - in which NetBackup XBSA objects may be created or deleted.
- NetBackup XBSA object retrieval transactions - in which NetBackup XBSA objects can only be queried and/or retrieved. This type of transaction provides no functional benefit for the calling XBSA application, and is only included for completeness.

The type of a transaction is established by the first create/delete/retrieve operation performed. Attempts to mix operations in a transaction result in a BSA_RC_INVALID_CALL_SEQUENCE error. The permissible call sequences are defined later in this chapter.

Once a transaction starts, many of the XBSA environment variables can no longer be reset. BSA_SERVICE_HOST, NBBSA_CLIENT_HOST, NBBSA_POLICY, and NBBSA_SCHEDULE cannot be modified within a transaction. If these need to be modified, the XBSA application must exit the transaction, make the variable changes, and start a new transaction.

Backup transaction

An XBSA application can create a NetBackup XBSA object in a backup transaction. The first BSACreateObject() call defines the backup transaction. The BSACreateObject() function takes as input an object descriptor that has all of the XBSA attributes of the object. After the BSACreateObject() call, the object's data is passed to NetBackup in buffers using a sequence of BSASendData() calls. When all data has been sent, the object is completed with a BSAEndData() call. Multiple objects may be created in one transaction, although BSAEndData() must be called before the next BSACreateObject() is called.

The NetBackup XBSA interface treats backup and archive transactions the same. The XBSA application performs any extra operations that can be associated with an archival. The XBSA application is also responsible for any other backup types such as an incremental backup. The NetBackup archive and incremental backups do not apply to the NetBackup XBSA interface. All of the information that is required to restore an object needs to be contained in the object descriptor or object data.

Within a backup transaction, query, delete, and restore operations are not allowed.

Restore transaction

The Restore transaction is similar to Backup transaction, except that the data flow is reversed. The restore transaction is defined by a call to BSAGetObject().

To restore an XBSA object, the NetBackup XBSA interface needs to know the copyId of that object. The copyId can be obtained from a catalog that is maintained

by the XBSA application or from a previous BSAQueryObject() call. Query operations can be mixed in with restore operations to get this data.

The BSAGetObject() call is used to initiate the restore of an object. It takes as input an object descriptor that contains the copyId of the object to be restored. Then, a series of BSAGetData() calls are used to get data for the object in buffers. The BSAEndData() call signals the end of getting data for the object. It is up to the XBSA application to recreate the object being restored using the object descriptor and data. When restoring multiple objects, the XBSA application must get all data for an object and call BSAEndData() before it calls BSAGetObject() to start restoring the next object.

Within a restore transaction, it is permissible to have BSAQueryObject() and BSAGetNextQueryObject() calls. This lets the XBSA application intermix restore operations with BSAQueryObject() and BSAGetNextQueryObject() calls to restore multiple objects within one transaction. Backup and delete operations are not allowed within a restore transaction.

It should be noted that the use of transactions for restore operations does not provide any functional benefit to the XBSA application but is required for completeness. If a restore is aborted with a call to BSAEndTxn() or BSATerminate() before the restore has completed, the NetBackup XBSA interface frees the NetBackup resources. It is up to the XBSA application to leave the object being restored in a consistent state.

Delete transaction

An XBSA application can delete a NetBackup XBSA object using the BSADeleteObject() call. BSADeleteObject() takes a copyId as a parameter and marks that object to be deleted. The actual delete of an object does not take place until the BSAEndTxn() call commits the transaction, so a query within a delete transaction can return an object to be deleted. An XBSA application can delete an NetBackup image that contains one or more objects using the NBBSADeleteImage() call. NBBSADeleteImage() takes a copyId of any one of the objects in the image as a parameter. Unlike BSADeleteObject(), the deletion of the image takes place during the NBBSADeleteImage() call. If objects were backed up to a tape device, the data is not deleted as part of this transaction. When all images on a tape have been deleted or expired, NetBackup frees the tape to be reused.

Within a delete transaction, it is permissible to embed BSAQueryObject() and BSAGetNextQueryObject() calls. This allows the XBSA application to intermix delete operations with BSAQueryObject() and BSAGetNextQueryObject() calls to delete multiple objects or images within one transaction. Backup and restore operations are not allowed within a delete transaction.

Note: For NetBackup XBSA Version 1.1.0, BSADeleteObject() has a limitation that there can only be one object in a NetBackup image for the delete to work. This means that when the object was created, it was the only object created in the transaction. If there are multiple objects, BSADeleteObject() returns a BSA_RC_SUCCESS status, but the object still exists. To delete a NetBackup image containing multiple objects, use NBBSADeleteImage().

NetBackup takes care of deleting objects with the retention period setting that is part of the configuration of a NetBackup schedule. In general, due to the way the data is stored on tape and other media, deleting individual objects has limited value.

Query transaction

An XBSA application can query for NetBackup XBSA objects that have been created in a query transaction. The BSAQueryObject() call is used to query the NetBackup catalog for NetBackup XBSA objects. Since retention of NetBackup XBSA objects is a function of NetBackup there is no guarantee that the call to BSAQueryObject() returns any objects.

The query is based on a subset of the object descriptor attributes, contained in a query descriptor. All fields in the query descriptor must be populated and the query searches for the objects that match all fields. Each of the fields has a wildcard or 'ANY' value that can be used. If you leave a field blank, it only matches the objects that also have blanks in that field.

The result of a query can return object descriptors, but never XBSA object data. If a query finds multiple object descriptors, BSAQueryObject() returns the first object descriptor and the remaining objects can be retrieved one at a time by using a succession of BSAGetNextQueryObject() calls.

It should be noted that the use of transactions for query operations does not provide any functional benefit to the XBSA application but it is required for completeness. As noted in the other transaction types, queries can be embedded in restore and delete transactions.

Media IDs transaction

An XBSA application can obtain the media IDs of a NetBackup image that contains one or more objects using the NBBSAGetMediaIds() call. NBBSAGetMediaIds() takes a copyId of any one of the objects in the image as a parameter. NBBSAGetMediaIds() returns a null character delimited text string that contains the media IDs associated with a NetBackup image in the buffer supplied as a parameter. The list of media IDs is terminated with an empty string. This is called a double null terminated string.

Examples are as follows:

- Image with two media IDs: `MediaId1\0MediaId2\0\0`
- Image with one media ID: `MediaId3\0\0`

Within a media ID transaction, you can embed BSAQueryObject() and BSAGetNextQueryObject() calls. This lets the XBSA application intermix the media ID operations with BSAQueryObject() and BSAGetNextQueryObject() calls to obtain the media IDs for multiple images within one transaction. Backup and restore operations are not allowed within a media ID transaction.

Creating a NetBackup XBSA application

This section contains information on initiating an XBSA session, using XBSA objects, logging, running an XBSA application in a clustered environment, and hints for getting the best performance out of the NetBackup XBSA interface.

Initiating a session

A session is initiated with a call to BSAInit(). One of the parameters of BSAInit() is the list of environment variables that is used to set up the XBSA environment between the XBSA application and the NetBackup XBSA interface. The only variable that is required by the NetBackup XBSA interface is BSA_API_VERSION. BSAInit() validates that the XBSA application uses a supported version. Other environmental variables can be included to increase flexibility of the application or to override values from the NetBackup configuration. But if these variables are not set, there are defaults from the configuration that are used.

Using these environment variables does not allow the XBSA application to bypass the NetBackup configuration, only to change from the default. All hosts, policies, schedules, and so on that are used must still be defined in the NetBackup configuration in order for the transactions to work. See the NetBackup System Administrator's Guide, Volume I, for more information on how to configure NetBackup.

The XBSA application should allow the XBSA environment variables to be set from run time values. These values can be obtained from parameters or from system environment variables. This allows the maximum flexibility for the application.

See “About How to run a NetBackup XBSA application” on page 73.

Some of the XBSA environment variables must be specified in the call to BSAInit() and cannot be changed within the session. Others can be set or modified within the session which gives the XBSA application maximum flexibility.

More information is available for the individual variables.

See “NetBackup XBSA environment” on page 24.

Modifying the XBSA environment within a session

The XBSA environment is created when the session is initiated. A couple of the variables, like BSA_API_VERSION and NBBSA_LOG_DIRECTORY, cannot be changed once the session has started. Many of the other variables can still be modified. If the XBSA application is going to set BSA_SERVICE_HOST, NBBSA_CLIENT_HOST, NBBSA_POLICY, or NBBSA_SCHEDULE, this needs to be done outside of a transaction, either before the first transaction or between transactions.

Once within a session, the XBSA environment can be updated with either NBBSASetEnv() or NBBSAUpdateEnv(). These are extensions to the XBSA specification. NBBSASetEnv() is used to set an individual XBSA environment variable and NBBSAUpdateEnv() updates the entire XBSA environment.

Session example

The following example sets up a session and begins a transaction. It sets up the XBSA environment, a BSA_ObjectOwner structure, and a BSA_SecurityToken. The security token is NULL because the NetBackup XBSA interface does not use this security method. The session is initiated by a BSAInit() call that returns a BSA_Handle. This handle is then used when a transaction begins and for all XBSA function calls within the session. Within the session, the XBSA environment is modified to change the NBBSA_CLIENT_HOST. Lastly a transaction is started.

```
BSA_Handle                  BsaHandle;
BSA_Obje ctOwner            BsaObjectOwner;
BSA_SecurityToken           *security_tokenPtr;
BSA_UInt32                  Size;
char                        *envx[3];
char                        ErrorString[512];
char                        msg[1024];
int                         status;

/ * Allocate memory for the XBSA environment variable array. */
envx[0] = malloc(40);
envx[1] = malloc(40);

/ * Populate the XBSA environment variables for this session.
  * Normally the BSA_SERVICE_HOST would not be hard coded like this but
  * would be retrieved via a parameter or environment variable.
  */
```

```
strcpy(envx[0], "BSA_API_VERSION=1.1.0");
strcpy(envx[1], "BSA_SERVICE_HOST=server_host");
envx[2] = NULL;

/ * The NetBackup XBSA Interface does not use the security token. */

security_tokenPtr = NULL;

/ * Populate the object owner structure. */

strcpy(BsaObjectOwner.bsa_ObjectOwner,"XBSA Client");
strcpy(BsaObjectOwner.app_ObjectOwner,"XBSA App");

/ * Initialize an XBSA session. */
status = BSAInit(&BsaHandle,NULL,&BsaObjectOwner,envx);
if (status != BSA_RC_SUCCESS) {
   Size = 512;
   NBBSAGetErrorString(status, &Size, ErrString);
   printf("ERROR: BSAInit failed with error: %s\n", ErrString);
   exit(status);
}

/ * Set the hostname of the client for the next transaction.  */
NBBSASetEnv(BsaHandle, "NBBSA_CLIENT_HOST", "client_host");

/ * Begin a transaction.  If it fails, terminate the session. */
status = BSABeginTxn(BsaHandle);
if (status != BSA_RC_SUCCESS) {
   Size = 512;
   NBBSAGetErrorString(status, &Size, ErrorString);
   sprintf(msg, "ERROR: BSABeginTxn failed with error: %s",
       ErrorString);
   NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
   BSATerminate(BsaHandle);
   exit(status);
}
```

Backup - creating an object

Once the application has started a transaction, it can start a backup. A backup transaction is identified by the first BSACreateObject() call. BSACreateObject() starts the process of backing up an object. Once the object has been created,

multiple BSASendData() calls are used to send the data associated with an object. This object is then completed with a BSAEndData() call.

The ability to pass data in buffers allows an XBSA application to use any buffering technique that is appropriate to ensure consistency or to improve performance. When data is passed in buffers, all the data for one object must be passed, in the proper sequence, before any other operation is started.

Creating an object

An object descriptor defines an XBSA object. The XBSA application defines the attributes of the object so that the application knows how to restore the object. For example, if the XBSA application wants to implement an incremental type of backup, sufficient information needs to be kept in the object descriptor to identify if the object is full or incremental and any other information that is required to restore the object.

The following fields of an object descriptor are user-defined and need to be defined by the XBSA application before the descriptor is passed to BSACreateObject().

See "Object descriptors" on page 18. for more definitions of the BSA_ObjectDescriptor.

The fields that are defined as strings can be empty strings, except for the pathName, which must have a valid path. The fields that are enumerations cannot have the ANY value. The estimatedSize field must have a value greater than zero if the object has data and zero if there is no data. It is good practice to have the estimated size field be as accurate as possible, but it does not affect how NetBackup stores the object.

The following are the required BSA.ObjectDescriptor fields:

```
objectOwner
   bsa_objectOwner
   app_objectOwner
objectName
   pathName
   objectSpaceName
copyType
resourceType
objectType
objectDescription
estimatedSize
objectInfo
```

The NetBackup XBSA interface populates the other fields in the object descriptor.

The other structure that is required before creating an object is the BSA_DataBlock32 structure. The structure does not need to be populated because BSACreateObject() populates the select fields with values that define how the data needs to be passed in buffers.

See "Buffers" on page 21. for more information.

Those are the two parameters to BSACreateObject(). The BSACreateObject() function creates the object and prepares NetBackup to be able to accept data. This includes mounting a tape if that is required. When BSACreateObject() has successfully created the object and returns, the object descriptor has the copyId field populated. This is the unique identifier that is associated with this object. If the XBSA application is going to keep any information about an object in an application catalog, this copyId should be a key value. It can be used to restore or delete this object.

There are four environmental variables that are created during BSACreateObject(). These are NBBSA_CLIENT_READ_TIMEOUT, NBBSA_MEDIA_MOUNT_TIMEOUT, NBBSA_MULTIPLEXING, and NBBSA_SERVER_BUFFSIZE. These variables are part of the NetBackup configuration and can be used to determine if the XBSA application is successful. The NBBSA_CLIENT_READ_TIMEOUT and NBBSA_MEDIA_MOUNT_TIMEOUT values can be reset by the XBSA application if it knows it needs to override the default NetBackup configuration.

NBBSA_CLIENT_READ_TIMEOUT is the amount of time, in seconds, the NetBackup server waits for data to be received. If the time between when the NetBackup server starts the backup and the time the transmission of data starts exceeds this time-out value, the backup job fails. This ensures that a hung or failed process on the client does not cause the job to wait, and take up resources, indefinitely. If the XBSA application knows that it takes longer than this to prepare the data to be sent, reset this value to a higher value.

NBBSA_MEDIA_MOUNT_TIMEOUT is the amount of time the NetBackup client waits for the media to be mounted. If the time between when the NetBackup server starts the backup and the time the media is mounted exceeds this time-out value, the XBSA interface returns a fail condition.

NBBSA_MULTIPLEXING is the number of streams that can be accepted by NetBackup. This value cannot be changed, but if the XBSA application is processing multiple streams, it should be evaluated to ensure that NetBackup accepts all of the streams that are being sent.

NBBSA_SERVER_BUFFSIZE is the size configured for NET_BUFF_SZ. This value cannot be changed but, if the XBSA application has the ability to modify the size of the buffers it uses, these could be modified to enhance performance of the transfer of data.

If everything is OK so far, data can be sent to the NetBackup XBSA interface by buffers by BSASendData(). The buffers are defined by the BSA_DataBlock32 structure. The key fields to set are the numBytes, which contains the number of bytes being sent, bufferLen, which contains the length of the buffer in bytes, and bufferPtr, which is a pointer to the buffer. The number of bytes must equal the buffer length except for the last buffer, which can be only partially full. BSASendData() can be called any number of times to pass all the data from an object.

Once all data has been sent, BSAEndData() must be called to signal to the NetBackup XBSA interface that the object is complete.

If multiple objects are to be created, this whole process can be repeated multiple times. The most efficient way to create multiple objects is to repeat this within one transaction. It is also possible to create multiple objects by creating one object per transaction and doing multiple transactions.

Once all objects for a transaction have been created, the transaction is completed with BSAEndTxn(). BSAEndTxn() can either commit or abort the transaction. If the transaction is aborted, all objects that were created in the transaction are not saved. If the transaction is committed, the object(s) are saved in the NetBackup catalog and can at a future point be restored. The BSATerminate() function also acts as an abort to the transaction.

NetBackup object ownership

Default behavior

When the NetBackup XBSA interface is used to create an object, by default the owner of the object is the logon user of the process that created the object. The default group of the object is also the logon user, not the primary group of the logon user, but the exact same name as the logon user name. The permissions of the file are set to 600, or ‘rw- - - - - - -’, which is read/write for owner and no access permissions for anyone else. This requires that the user restoring an object be an administrator or the same user that created the object. The XBSA objectOwner fields are saved in the NetBackup catalog with the object, but they are kept as attributes of the object and are not used for security purposes.

Ownership options

Using the XBSA environmental variables NBBSA_USE_OBJECT_OWNER, NBBSA_USE_OBJECT_GROUP, NBBSA_OBJECT_OWNER, and NBBSA_GROUP_OWNER, an agent can change the default owner. These variables allow the XBSA agent to be able to specify who owns the objects.

Note: Specifying object ownership only works when creating objects using BSACreateObject(). Accessing the objects by BSAQueryObject() and BSAGetObject() is dependent on the logon process having permissions to access the objects. So if user_Y creates an object with an object owner of user_X, then user_X or an administrator (root) can access and restore the object, but user_Y cannot.

Object owner

To specify the owner of an object, the XBSA environment variable NBBSA_USE_OBJECT_OWNER needs to be set. There are four values that this variable can be set to. These values are defined in `nbbsa.h`.

```
/*
 * XBSA values to use to define how to specify NetBackup object ownership
 */
#define VxLOGIN_USER  0 /* Default, owner/group field is set to the logon user */
#define VxLOGIN_GROUP 1 /* group field is set to the primary group of the logon user */
#define VxBSA_OWNER   2 /* owner/group field is set to                                  \
objectDescriptor->objectOwner.bsa_ObjectOwner  */
#define VxAPP_OWNER   3 /* owner/group field is set to                                  \
objectDescriptor->objectOwner.app_ObjectOwner  */
#define VxENV_OWNER   4 /* owner/group field is set to value of                         \
NBBSA_OBJECT_OWNER/NBBSA_OBJECT_GROUP */
```

VxLOGIN_USER is the default behavior that you would get if the NBBSA_USE_OBJECT_OWNER variable wasn't set.

VxLOGIN_GROUP does not apply to object ownership.

VxBSA_OWNER sets the object owner to the value stored in the objectDescriptor field objectOwner.bsa_ObjectOwner. The value in the bsa_ObjectOwner field must be a valid user name without any spaces in the name. The value in objectOwner.bsa_ObjectOwner is still stored as an attribute of the object and a query must correctly specify this field in the query descriptor to successfully find the object.

VxAPP_OWNER sets the object owner to the value stored in the objectDescriptor field objectOwner.app_ObjectOwner. The value in the app_ObjectOwner field must be a valid user name without any spaces in the name. The value in objectOwner.app_ObjectOwner is still stored as an attribute of the object and a query needs to correctly specify this field in the query descriptor to successfully find the object.

VxENV_OWNER sets the object owner to the value of the XBSA environmental variable NBBSA_OBJECT_OWNER. The value stored in the

NBBSA_OBJECT_OWNER must be a valid user name without any spaces in the name.

The variables NBBSA_USE_OBJECT_OWNER and NBBSA_OBJECT_OWNER can be changed within a transaction so that an XBSA agent can set different ownerships of each object in a transaction.

Object group

An XBSA agent can also change the group ownership of an object. When the group ownership is set by one of these options, other than the default, the permissions on the object are set to 660, or 'rw - rw- - - -', which is read/write for owner and group. This allows any user in the specified group to access and restore the object.

To specify the group of an object, the XBSA environment variable NBBSA_USE_OBJECT_GROUP needs to be set. There are five values that this variable can be set to. These values are defined in nbbsa.h.

```
/*
 * XBSA values to use to define how to specify NetBackup object ownership
 */
#define VxLOGIN_USER 0 /* Default, owner/group field is set to the logon user  */
#define VxLOGIN_GROUP 1 /* group field is set to the primary group of the logon user */
#define VxBSA_OWNER 2 /* owner/group field is set to                                    \
objectDescriptor->objectOwner.bsa_ObjectOwner  */
#define VxAPP_OWNER 3 /* owner/group field is set to                                    \
objectDescriptor->objectOwner.app_ObjectOwner  */
#define VxENV_OWNER 4 /* owner/group field is set to value of                           \
NBBSA_OBJECT_OWNER/NBBSA_OBJECT_GROUP */
```

VxLOGIN_USER is the default behavior that you would get if the NBBSA_USE_OBJECT_GROUP variable was not set. The group name is the same name as the owner field, whether that is the logon user or a user name defined by one of the other options, and the permissions of the object will be 600, owner read/write only.

VxLOGIN_GROUP sets the group field to the primary group of the logon user.

VxBSA_OWNER sets the object group to the value stored in the objectDescriptor field objectOwner.bsa_ObjectOwner. The value in the bsa_ObjectOwner field must be a valid user name without any spaces in the name. The value in objectOwner.bsa_ObjectOwner still is stored as an attribute of the object and a query must correctly specify this field in the query descriptor to successfully find the object.

VxAPP_OWNER sets the object group to the value stored in the objectDescriptor field objectOwner.app_ObjectOwner. The value in the app_ObjectOwner field must be a valid user name without any spaces in the name. The value in

objectOwner.app_ObjectOwner is still stored as an attribute of the object and a query must correctly specify this field in the query descriptor to successfully find the object.

VxENV_OWNER sets the object group to the value of the XBSA environmental variable NBBSA_OBJECT_GROUP. The value stored in the NBBSA_OBJECT_GROUP must be a valid user name without any spaces in the name.

The variables NBBSA_USE_OBJECT_GROUP and NBBSA_OBJECT_GROUP can be changed within a transaction so that an XBSA agent can set different group ownerships of each object in a transaction.

Creating an empty object

You can create an XBSA object without any associated data. This is created in much the same way as an object with data with two differences. The estimatedSize.left and estimatedSize.right fields need to be zero so that the NetBackup XBSA interface knows that the object is going to be empty. After the BSACreateObject() call, the XBSA application calls BSAEndData() to end the object. If estimatedSize is set to zero and BSASendData() is called, this results in an error.

Backup example

The following example goes through the process of creating an object. It is assumed the transaction has already been started). The BSA_ObjectDescriptor is populated with the values that are associated with the object. Then the DataBlock32 structure is created to receive any restrictions put on the data by the NetBackup Interface. BSACreateObject() is then called to create the object and start the backup process. Once the object is created, this example sends one buffer of data with the BSASendData() call. After the last BSASendData() call, the object is completed with a BSAEndTxn(), which commits the object.

This example only creates one object and only sends one buffer of data. In general, objects take multiple buffers and a transaction can create multiple objects.

```
BSA_Handle                      BsaHandle;
BSA_ObjectOwner                 BsaObjectOwner;
BSA_SecurityToken               *security_tokenPtr;
        BSA_DataBlock32         *data_block;
BSA_ObjectDescriptor            *object_desc;
BSA_UInt32                      DataBuffSz;
BSA_UInt32                      Size;
char                            *envx[5];
char                            DataBuff[512];
char                            ErrorString[512];
```

```
char                                msg[1024];
int                                 status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/ * Populate object descriptor of the first object to be backed up. */

object_desc = (BSA_ObjectDescriptor *)malloc(sizeof(BSA_ObjectDescriptor));

strcpy(object_desc->objectOwner.bsa_ObjectOwner, "XBSA Client");
strcpy(object_desc->objectOwner.app_ObjectOwner, "XBSA App");
strcpy(object_desc->objectName.pathName, "/xbsa/sample/object1");
strcpy(object_desc->objectName.objectSpaceName, "");
strcpy(object_desc->resourceType, "Sample");
strcpy(object_desc->objectDescription,"Sample description of Obj 1");
strcpy(object_desc->objectInfo,"Object 1");
object_desc->copyType = BSA_CopyType_BACKUP;
object_desc->estimatedSize.left = 0;
object_desc->estimatedSize.right = 100;
object_desc->objectType = BSA_ObjectType_FILE;

/ *  Initialize the BSA_DataBlock32 structure. */

data_block = (BSA_DataBlock32 *)malloc(sizeof(BSA_DataBlock32));
memset(data_block, 0x00, sizeof(BSA_DataBlock32));
/ * Create sample object. If object cannot be created, terminate session. */

status = BSACreateObject(BsaHandle, object_desc, data_block);
if (status == BSA_RC_SUCCESS) {
    printf("copyId: %d - %d\n", object_desc->copyId.left, object_desc->copyId.right);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSACreateObject failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}
```

```
/ * For the purposes of this sample, we will assume that the data in the *
  * DataBuff buffer has been populated from reading the data from the object *
  * being backed up. */

strcpy(DataBuff, "This is the sample data that is contained in the sample object that
is being backed up for the purposes of showing how data can be backed up and
restored.");
DataBuffSz = strlen(DataBuff);

/ * BSACreateObject sets values in the BSA_DataBlock32 to indicate    *
  * header and trailer requirements. The NetBackup implementation has *
  * no such requirements and are not checked here.  Set the other     *
  * fields of the data_block for the BSASendData call.              */

data_block->bufferLen = 512;
data_block->bufferPtr = DataBuff;
data_block->numBytes = DataBuffSz;

/ * Send the data to be backed up.  In normal implementations, BSASendData    *
  * would be in a loop reading the data into the buffer and then sending it   *
  * until all the data of the object has been sent.                        */

status = BSASendData(BsaHandle, data_block);
if (status == BSA_RC_SUCCESS) {
    printf("Bytes backed up: %d\n", data_block->numBytes);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSASendData failed with error: %s\n", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ *  All data has been sent for the object.  */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndData failed with error: %s", ErrorString);
```

```
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ *  End the backup transaction and commit the object.  */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Query - finding an object descriptor

The XBSA application can query the NetBackup XBSA interface for XBSA objects that have been created. The BSAQueryObject() call is used to query the NetBackup catalog for these objects. The query is based on a subset of the object descriptor attributes, contained in a query descriptor. If the result of the query is multiple object descriptors, BSAQueryObject() returns the first (most recent) object and the rest can be retrieved one object descriptor at a time by using a succession of BSAGetNextQueryObject() calls.

Querying for an object

When querying for an object, the object attributes that the XBSA application is querying for are contained in a query descriptor. This query descriptor is made up of strings and enumerations. They are evaluated against the objects stored in the NetBackup catalog for objects that match all fields. Each field of the query descriptor must be populated. If a string field is populated with an empty string or NULL, it only matches objects that also have an empty string for that field. Wildcards and 'ANY' enumerations allow the XBSA application to search for objects that have some fields that are unknown.

There are two fields that are not part of the XBSA specifications but can be very useful. The createTime_from and createTime_to fields limit the search to the time period between these dates. These are optional fields, the default is to search all objects, but can greatly speed up the search when the NetBackup catalog is very large.

When doing the query, the XBSA application only returns objects that are owned by the logon user running the query, unless that user is a root admin. NetBackup XBSA Version 1.1.0 uses the logon user as the object owner. The objectOwner field is considered an attribute and is not used for security.

The query, by default, also only returns objects that were created on the hostname from which the query is being run. If the XBSA application needs to find an object that was created from a different host, the NBBSA_CLIENT_HOST environment variable must be set to the hostname from which the object was created. This variable can only be set before a transaction begins. If the application is looking for objects from multiple hosts, the application needs to do queries in separate transactions.

Query example

Here is an example of a query. It starts with populating a query descriptor, which identifies what objects are being searched for. Then it makes the initial query

```
BSA_Handle                    BsaHandle;
BSA_ObjectOwner               BsaObjectOwner;
BSA_SecurityToken             *security_tokenPtr;
BSA_QueryDescriptor           *query_desc;
BSA_ObjectDescriptor          *object_desc;
BSA_UInt32                    Size;
char                          *envx[3];
char                          ErrorString[512];
char                          msg[1024];
int                           status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/ *  Populate the query descriptor of the object to be searched for. */

query_desc = (BSA_QueryDescriptor *)malloc(sizeof(BSA_QueryDescriptor));
memset(query_desc, 0x00, sizeof(BSA_QueryDescriptor);

query_desc->copyType = BSA_CopyType_BACKUP;
query_desc->objectType = BSA_ObjectType_FILE;
query_desc->objectStatus = BSA_ObjectStatus_ANY;
strcpy(query_desc->objectOwner.bsa_ObjectOwner, "XBSA Client");
```

```
strcpy(query_desc->objectOwner.app_ObjectOwner, "XBSA App");
strcpy(query_desc->objectName.pathName, "/xbsa/sample/object1");
strcpy(query_desc->objectName.objectSpaceName, "");

object_desc = (BSA_ObjectDescriptor *)malloc(sizeof(BSA_ObjectDescriptor));

/ * Begin searching for objects matching the query criteria.  BSAQueryObject()    *
  * returns the first (most recent) object found.                                 */

status = BSAQueryObject(BsaHandle, query_desc, object_desc);
if (status == BSA_RC_SUCCESS) {
    printf("copyId: %d - %d\n", object_desc->copyId.left, object_desc->copyId.right);
} else if (status == BSA_RC_NO_MATCH) {
    sprintf(msg, "WARNING: BSAQueryObject() did not find an object matching the
       query");
    NBBSALogMsg(BsaHandle, MSWARNING, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAQueryObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
}

/ * Continue searching for other objects which match the query criteria.  *
  * BSAGetNextQueryObject() should return BSA_RC_NO_MORE_DATA when there  *
  * are not more objects.                                                 */

while ((status = BSAGetNextQueryObject(BsaHandle, object_desc)) == BSA_RC_SUCCESS) {
    printf("CopyId: %d.%d\n", object_desc->copyId.left, object_desc->copyId.right);
}
if (status != BSA_RC_NO_MORE_DATA) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAGetNextQueryObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
}
```

```
/ * End the query transaction.  BSA_Vote_COMMIT and BSA_Vote_ABORT are *
  * equivalent as there is nothing to commit or abort.                 */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Restore - retrieving an object's data

Another type of transaction is a restore transaction. A restore transaction is identified by the first BSAGetObject() call. A difference from a backup transaction is that there can also be BSAQueryObject() calls within a restore transaction, which is useful to get the object descriptor of the object the XBSA application is restoring. BSAGetObject() starts the process of retrieving an object. Once the object has been retrieved, multiple BSAGetData() calls are used to retrieve the data associated with an object. The last BSAGetData() call returns BSA_NO_MORE_DATA that signals that the NetBackup XBSA interface has completed sending the data. The BSAEndData() call then releases all of the resources.

Restoring an object

When restoring an XBSA object, the logon user must be the owner of the XBSA object or a root admin. (The owner of an object is the logon user of the process that created the object.) If a different user tries to restore the object, the NetBackup XBSA interface returns a BSA_RC_OBJECT_NOT_FOUND error. This error could also be returned if the host on which the restore is being done is different from the host which backed up the object.

See "Redirected restore to a different client" on page 53.

The XBSA application is responsible for recreating the object. The NetBackup XBSA interface sends a stream of data to the XBSA application. It is up to the XBSA application to ensure that the correct permissions exist for restoring the object, recreating all attributes, and so on. If any of these attributes are stored in the object descriptor of the XBSA object, the object descriptor needs to be retrieved with a BSAQueryObject() call. The call to BSAGetObject() does not populate the object attributes.

To restore an XBSA object, the NetBackup XBSA interface needs to have an object descriptor that contains the copyId of the object being restored. This copyId can be obtained from either a query process or from information stored by the XBSA application. It is permissible to mix query operations in a restore transaction.

The other structure that is required before restoring an object is the BSA_DataBlock32 structure. The structure does not need to be populated as BSAGetObject() populates select fields with values that define how the data buffers are used.

See "Buffers" on page 21.

The restore is initiated with a call to BSAGetObject() with this object descriptor and data block as parameters. This function starts the process of retrieving the object. If BSAGetObject() returns with good status, BSAGetData() can retrieve the object data from the NetBackup XBSA interface by buffers. The buffers are defined by the BSA_DataBlock32 structure. It is the responsibility of the XBSA application to allocate the buffers. BSAGetObject() fills the buffers with data and sets the numBytes field of the BSA_DataBlock32 with the number of bytes in the buffer. When the last buffer of data for the object has been passed, BSAGetObject() returns BSA_NO_MORE_DATA. BSAEndData() should then be called to signal to the NetBackup XBSA interface that the object is restored and that it can free up the resources. The NetBackup XBSA interface requires that all data for an object is retrieved or the return status of the NetBackup server would be an error status. This does not affect the XBSA application, but may affect how a user of the application interprets the results of the restore.

After the object(s) have been restored, the transaction should be closed. From the NetBackup XBSA interface point of view, a committed or aborted transactions are handled the same, as there is nothing for NetBackup to commit.

Redirected restore to a different client

One specific type of restore that deserves special notice is what is considered a redirected restore to a different client. An XBSA object is stored within NetBackup with a specific client from which it was backed up. The default is to assume that the object is being restored to the same client. If the hostname that is initiating the restore is different from the hostname on which the object was backed up, the NBBSA_CLIENT_HOST environment variable needs to be set.

The NBBSA_CLIENT_HOST must be set, before entering the transaction, to the hostname on which the object was backed up. If this variable has not been specified, the NetBackup XBSA interface cannot find the object.

Restore example

Here is an example of a restore. It assumes that the object descriptor has been populated with the copyId of the object either from a query or the XBSA application having stored this information.

```
BSA_Handle                      BsaHandle;
BSA_ObjectOwner                 BsaObjectOwner;
BSA_SecurityToken               *security_tokenPtr;
BSA_DataBlock32                 *data_block;
BSA_UInt32                      EnvBufSz = 512;
BSA_ObjectDescriptor            *object_desc;
BSA_QueryDescriptor             *query_desc;
BSA_UInt32                      Size;
char                            *envx[3];
char                            EnvBuf[512];
char                            ErrorString[512];
char                            msg[1024];
char                            *restore_location;
int                             total_bytes = 0;
int                             status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/ *  Get the object.  */

data_block = (BSA_DataBlock32 *)malloc(sizeof(BSA_DataBlock32));

status = BSAGetObject(BsaHandle, object_desc, data_block);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAQueryObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * The application is responsible for recreating the file or other object   *
```

```
/ * type that is being restored using the information that is stored in the  *
/ * object_descriptor.  This sample prints the results to the screen.        */

restore_location = (char *)malloc((EnvBufSz + 1) * sizeof(char));
memset(restore_location, 0x00, EnvBufSz + 1);

/ * Initialize the data_block structure.  */

data_block->bufferLen = EnvBufSz;
data_block->bufferPtr = EnvBuf;
memset(data_block->bufferPtr, 0x00, EnvBufSz);

/ * Read data until the end of data. */

while ((status = BSAGetData(BsaHandle, data_block)) == BSA_RC_SUCCESS) {

    / * Move the retrieved data to where it is to be restored to and  *
      * reset the data_block buffer.                                   */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("%s", restore_location);

    memset(restore_location, 0x00, EnvBufSz + 1);
    memset(data_block->bufferPtr, 0x00, EnvBufSz);
}
if (status == BSA_RC_NO_MORE_DATA) {

    / * The last BSAGetData() that returns BSA_RC_NO_MORE_DATA may have data  *
      * in the buffer.                                                        */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("%s\n", restore_location);
    printf("Total bytes retrieved: %d\n", total_bytes);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAGetData() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
```

```
    BSATerminate(BsaHandle);
    exit(status);
}

/ * Done retrieving data. */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndData() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * End the restore transaction.  BSA_Vote_COMMIT and BSA_Vote_ABORT are         *
/ * equivalent as there is nothing to commit or abort for a restore transaction. */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Multiple object restore

If multiple objects are going to be restored in one session or transaction, the XBSA agent should consider using the NBBSAGetMultipleObjects function call. This is a NetBackup extension to the XBSA interface to optimize the retrieval of objects in a NetBackup environment. This is especially useful when retrieving many small objects.

The reason this provides a performance improvement is that each NetBackup restore operation creates a NetBackup job, which acquires resources and then frees them up when the job is complete. Each BSAGetObject call translates into one NetBackup job. The initial time required to start a NetBackup job and establish communication are minimal, especially when compared to the time to transfer large amounts of data. But if the objects are small and numerous, this overhead per object

is noticeable. It is also possible on heavily loaded NetBackup systems that successive BSAGetObject calls may get queued up behind other NetBackup jobs and resource requests. Any of these could cause performance issues if the separate objects are all part of one restore.

To address this behavior of NetBackup, we have added a multiple object restore interface to the XBSA interface. This is an extension of the XBSA specification to enhance performance for the NetBackup XBSA applications. The use of this interface is not required and does not provide functionality on objects that are not available through the interfaces defined by XBSA.

Requirements

- To do a multiple object restore, the XBSA application needs to have created the objects in ways that allow this and there are restrictions on how the objects can be retrieved.
- All of the objects that are to be restored within a multiple object restore must be part of the same NetBackup image, which means that the objects were created in one transaction. This can be verified by checking that each object being restored has the same copyId.right.
- The objects must be retrieved in the same order that they were created. Some objects in the image can be skipped, but the objects being retrieved must be ordered in a way that do not cause the media to have to position backwards. The ordering of objects can be determined by verifying that the copyId.left for each object is in ascending order.
- While not all objects in an image need to be retrieved, all objects in the list created by NBBSAAddToMultiObjectRestoreList must be retrieved in the order in which they are listed. Objects cannot be skipped or added.
- Each object in the list is retrieved with BSAGetObject followed by successive BSAGetData calls to retrieve all of the data. All of the data for an object must be retrieved before moving on to the next object.

Functions and use

There are three new functions provided as part of the XBSA interface that can be used to do multiple object restores.

- NBBSAAddToMultiObjectRestoreList takes an object descriptor and it to a descriptor list. This function is called for each object that is to be retrieved as part of one restore job. It is highly recommended to use this function to create the list because it allows the XBSA interface to be in charge of memory allocation and deletion.
- NBBSAGetMultipleObjects starts the multiple object restore job. It takes the list of descriptors built by NBBSAAddToMultiObjectRestoreList and submits a request to restore all objects.

- NBBSAEndGetMultipleObjects ends the multiple object restore job. This function cleans up the memory from the object list and allows the application to COMMIT or ABORT the restore, which has no real effect on the data.

The process is very similar to the single object restores. First, all objectDescriptors to be retrieved are added to a list using the NBBSAAddToMultiObjectRestoreList. The objectDescriptors can be generated from BSAQueryObject or populated by the application. Once the list is ready, a call to NBBSAGetMultipleObjects initiates the restore process. Then, each object is retrieved using the standard BSAGetObject, BSAGetData, and BSAEndData function calls. The difference is that BSAGetObject knows it is part of a larger restore job. After all objects have been retrieved, NBBSAEndGetMultipleObjects is called to end the restore process. The transaction can then be ended. If an object is skipped or not all data is retrieved, the entire job fails.

Multiple object restore example

Here is an example of a multiple object restore. Examples of BSAQueryObject and BSAGetObject are included elsewhere in this document, so this example bypasses some of the error handling associated with those calls.

```
BSA_Handle                  BsaHandle;
BSA_ObjectOwner             BsaObjectOwner;
BSA_SecurityToken           *security_tokenPtr;
BSA_DataBlock32             *data_block;
BSA_QueryDescriptor         *query_desc;
BSA_ObjectDescriptor        *object_desc;
BSA_ObjectDescriptor        *object_desc_current;
NBBSA_DESCRIPT_LIST         *object_list = NULL;
NBBSA_DESCRIPT_LIST         *object_list_current;
BSA_UInt32                  EnvBufSz = 512;
BSA_UInt32                  Size;
char                        *envx[3];
char                        EnvBuf[512];
char                        ErrorString[512];
char                        msg[1024];
char                        *restore_location;
int                         total_bytes = 0;
int                         status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
```

```
BSABeginTxn(BsaHandle);

/* Populate the query descriptor of the object to be searched for. */

query_desc = (BSA_QueryDescriptor *)malloc(sizeof(BSA_QueryDescriptor));
object_desc = (BSA_ObjectDescriptor *)malloc(sizeof(BSA_ObjectDescriptor));
data_block = (BSA_DataBlock32 *)malloc(sizeof(BSA_DataBlock32));

query_desc->copyType = BSA_CopyType_BACKUP;
query_desc->objectType = BSA_ObjectType_FILE;
query_desc->objectStatus = BSA_ObjectStatus_MOST_RECENT;
strcpy(query_desc->objectOwner.bsa_ObjectOwner, "BSA Client");
strcpy(query_desc->objectOwner.app_ObjectOwner, "BSA App");
strcpy(query_desc->objectName.pathName, "/xbsa/sample/object1");
strcpy(query_desc->objectName.objectSpaceName, "");

/* Search for an object matching the query criteria.  */

status = BSAQueryObject(BsaHandle, query_desc, object_desc);
if (status != BSA_RC_SUCCESS) {
   /* handle error condition */
}

/* Start building the objectList by adding the object descriptor to the list.  */

status = NBBSAAddToMultiObjectRestoreList(BsaHandle, &object_list, object_desc);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAAddToMultiObjectRestoreList() failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, " Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
    }

/* Search for a second object.  */

strcpy(query_desc->objectName.pathName, "/xbsa/sample/object2");
status = BSAQueryObject(BsaHandle, query_desc, object_desc);
if (status != BSA_RC_SUCCESS) {
   /* handle error condition */
}
```

```
/* Add the second object descriptor to the objectList.  */

status = NBBSAAddToMultiObjectRestoreList(BsaHandle, &object_list, object_desc);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAAddToMultiObjectRestoreList() failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, " Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}

/* Start the multiple object restore by passing in the object list. The object list
 * will be evaluated and the restore job will be started.
 */

status = NBBSAGetMultipleObjects(BsaHandle, object_list);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAGetMultipleObjects () failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, "Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}

/* Create a pointer to the object list in order to keep track of the current object
 * being restored.  A list created by the application could also be used.
 * Point the object descriptor at the first object
 */
object_list_current = object_list;
object_desc_current = object_list_current->Descriptor;

/*  Get the first object.  */

status = BSAGetObject(BsaHandle, object_desc_current, data_block);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}
```

```
restore_location = (char *)malloc((EnvBufSz + 1) * sizeof(char));
memset(restore_location, 0x00, EnvBufSz + 1);

data_block->bufferLen = EnvBufSz;
data_block->bufferPtr = EnvBuf;
memset(data_block->bufferPtr, 0x00, EnvBufSz);

/* Read data until the end of data. */

while ((status = BSAGetData(BsaHandle, data_block)) == BSA_RC_SUCCESS) {

    /* Move the retrieved data to where it is to be restored to and  *
     * reset the data_block buffer.                                  */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    memset(restore_location, 0x00, EnvBufSz + 1);
    memset(data_block->bufferPtr, 0x00, EnvBufSz);
}
if (status == BSA_RC_NO_MORE_DATA) {

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("Total bytes retrieved: %d\n", total_bytes);
} else {
    /* handle error condition */
}

/* Done retrieving data for the first object. */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}

/* Set the object descriptor to the next object in the list.  */
object_list_current = object_list_current->next;
if (object_list_current == NULL) {
    /* handle end of objects condition */
}
```

```
object_desc_current = object_list_current->Descriptor;
if (object_desc_current == NULL) {
    /* handle error condition */
}

/*  Get the next object.  */

status = BSAGetObject(BsaHandle, object_desc_current, data_block);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}

restore_location = (char *)malloc((EnvBufSz + 1) * sizeof(char));
memset(restore_location, 0x00, EnvBufSz + 1);

data_block->bufferLen = EnvBufSz;
data_block->bufferPtr = EnvBuf;
memset(data_block->bufferPtr, 0x00, EnvBufSz);

/* Read data until the end of data. */

while ((status = BSAGetData(BsaHandle, data_block)) == BSA_RC_SUCCESS) {

    /* Move the retrieved data to where it is to be restored to and  *
     * reset the data_block buffer.                                  */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    memset(restore_location, 0x00, EnvBufSz + 1);
    memset(data_block->bufferPtr, 0x00, EnvBufSz);
}
if (status == BSA_RC_NO_MORE_DATA) {

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("Total bytes retrieved: %d\n", total_bytes);
} else {
    /* handle error condition */
}

/* Done retrieving data for the second object. */
```

```
status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}

/* End the multiple object restore transaction. Set any references to objects
 * in the object list to NULL as the memory associated to the list has been freed.
 */

status = NBBSAEndGetMultipleObjects(BsaHandle, BSA_Vote_COMMIT, object_list);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAEndGetMultipleObjects() failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, "Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}
object_list_current = NULL;
object_desc_current = NULL;

/* End the restore transaction. BSA_Vote_COMMIT and BSA_Vote_ABORT are
 * equivalent as there is nothing to commit or abort for a restore transaction.
 */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, " Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Delete - deleting an object or image

Deleting a NetBackup XBSA object is done with the BSADeleteObject() function. BSADeleteObject() does not always delete the object specified, even if it returns a success status. The only objects that can be deleted using BSADeleteObject() are objects in which there was only one object created per transaction.

Based on those limitations, the BSADeleteObject() function is pretty straightforward. It takes a copyId as its parameter and deletes this object. Multiple objects can be deleted in one transaction and it is permissible to have query operations within a delete transaction. The object is not deleted until the transaction is committed, so these query operations in a delete transaction can return a deleted object.

Deleting a NetBackup image is done with the NBBSADeleteImage() function. NBBSADeleteImage() deletes an entire image that can contain one or more objects created per transaction.

The NBBSADeleteImage() function takes a copyId of any object contained in the image as its parameter and deletes the entire image. Multiple images can be deleted in one transaction and it is permissible to have query operations within a delete transaction. Unlike BSADeleteObject(), the deletion of the image takes place during the NBBSADeleteImage() call.

Note: Deletion only removes an entry from the NetBackup catalog and not from the storage media. NetBackup allows media to be imported to recreate all of the images that are stored on them, and doing so may cause a deleted object or image to reappear.

Delete example

This delete example is very simple. It assumes that the copyId has been populated from a previous query or from information stored by the XBSA application. It then deletes one object and commits the transaction that does the delete of the object. In another transaction, an entire image is deleted.

```
BSA_Handle                      BsaHandle;
BSA_ObjectOwner                 BsaObjectOwner;
BSA_SecurityToken               *security_tokenPtr;
BSA_ObjectDescriptor            *object_desc;
BSA_ObjectDescriptor            *object_desc2;
BSA_QueryDescriptor             *query_desc;
BSA_UInt32                      Size;
char                            *envx[3];
char                            ErrorString[512];
char                            msg[1024];
int                             status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
```

```
BSABeginTxn(BsaHandle);

/ * Delete the object from NetBackup. */

status = BSADeleteObject(BsaHandle, object_desc->copyId);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSADeleteObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Delete");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * End the delete transaction, commit will delete the object */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Delete");
    BSATerminate(BsaHandle);
    exit(status);
}

BSABeginTxn(BsaHandle);

/* Delete an image from NetBackup. It may contain multiple objects. */

status = NBBSADeleteImage(BsaHandle, object_desc2->copyId);
if (status != BSA_RC_SUCCESS) {
   Size = 512;
   NBBSAGetErrorString(statuGetSs, &Size, ErrorString);
   sprintf(msg, "ERROR: NBBSADeleteImage() failed with error: %s", ErrorString);
   NBBSALogMsg(BsaHandle, MSERROR, msg, "Delete Image");
   BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
   BSATerminate(BsaHandle);
   exit(status);
}

/* The deletion of the image has occurred during the NBBSADeleteImage() call */
```

```
status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
   Size = 512;
   NBBSAGetErrorString(status, &Size, ErrorString);
   sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
   NBBSALogMsg(BsaHandle, MSERROR, msg, "Delete Image");
   BSATerminate(BsaHandle);
   exit(status);
}
```

Media IDs - obtaining media IDs

Obtaining the Media IDs for a NetBackup image is done with the NBBSAGetMediaIds() function. NBBSAGetMediaIds () gets all of the Media IDs associated with an image that contains one or more objects. The NBBSAGetMediaIds () function takes a copyId of any object contained in the image as one of its parameters. The Media IDs for multiple images can be obtained in one transaction and you can have query operations within a Media ID transaction.

Media ID example

This Media ID example assumes that the copyId has been populated from a previous query or from information stored by the XBSA application.

```
BSA_Handle              BsaHandle;
BSA_ObjectOwner         BsaObjectOwner;
BSA_SecurityToken       *security_tokenPtr;
BSA_ObjectDescriptor    *object_desc;
BSA_QueryDescriptor     *query_desc;
BSA_UInt32              Size;
char                    *envx[3];
char                    ErrorString[512];
char                    msg[1024];
int                     status;
char                    buffer[100];
char                    *p;
BSA_UInt32              buffer_len;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);
```

```
buffer_len = 100;
status = NBBSAGetMediaIds(BsaHandle, object_desc->copyId, &buffer_len, buffer);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAGetMediaIds() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Media Ids");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

p = buffer;
while (*p != '\0') {
    printf("Media Id = %s\n", p);
p = p + (strlen(p) + 1);
}
printf("Buffer size used = %d\n", str_len);

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Media Ids");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
}
```

Logging and NetBackup

NetBackup has a log directory that contains the debug logs for the various processes that make up the NetBackup server and/or client. There is a configurable verbose level that controls how much information is logged to these debug logs. This verbose level is a value from 0 to 5, with 0 indicating minimal logging and 5 being debug. These logs are used by NetBackup support to help solve customer problems. The log directory is located at `/usr/openv/netbackup/logs` on UNIX systems and *`install directory`*`/Veritas/NetBackup/logs` on Windows. Within this directory are directories for the different processes such as bprd, bpbrm, and so on. One log file gets created for each day, and NetBackup automatically cleans up old files from this directory. The NetBackup XBSA interface by default logs to the directory `exten_client`.

With NetBackup 10.0, some of the NetBackup services use Unified Logging (VxUL). Those include scheduler components: `nbjm`, `nbpem` and `nbrb`. For more details on VxUL, please see the NetBackup Troubleshooting Guide and the NetBackup Logging Reference Guide. The NetBackup XBSA interface does not currently use VxUL.

The NetBackup XBSA interface allows the XBSA application to log in a manner consistent with other NetBackup processes. By using the NBBSALogMsg() function, the XBSA application logs messages to the same file as the NetBackup XBSA interface. This can cause some confusion for the developer at first, especially at high debug levels, but lets the application see what causes the errors and it could help NetBackup support see what the XBSA application is doing. The log messages contain a timestamp along with the process ID, which is useful when there are multiple processes going at once.

The log message also contains a debug level. The different error levels used by NetBackup are defined in nbbsa.h. One of these values should be used in the msgType parameter of NBBSALogMsg(). While there are no hard definitions of when to use each of these values, using these values may help if NetBackup support or engineering is ever involved in looking at a debug log.

```
#define MSINFO          4
#define MSWARNING       8
#define MSERROR        16
#define MSCRITICAL     32
```

The XBSA application is not required to log information to the NetBackup logs. If the XBSA application is the backup portion of another application or database, it may make more sense to log information to a place consistent with the rest of the application.

Client in a cluster

Running an XBSA application in a clustered environment is a valid mode of operation. The key thing about running in a cluster is to ensure that the client name used when an object is created is the same as the client name used when the object is being queried or retrieved. To ensure that the same client name is used, the XBSA application should use the virtual name of the clients in the cluster. The best way to do this would be to use the NBBSA_CLIENT_HOST variable and set it to the virtual name for both creating and retrieving an object. The virtual name needs to be the client name that is configured in the NetBackup policy. Another option is to configure the virtual name as the default NetBackup client name. Configuring in this way causes other NetBackup jobs, such as the file system backups, to use this virtual name also, which may not be desired. If neither of these options is used by the XBSA application, the XBSA interface uses the default client name, which is

the physical address of the client. The objects are created successfully, but if the query or retrieval is done from a different node in the cluster, the object is not found.

Performance considerations

For the most part, the performance of the NetBackup XBSA interface with the XBSA application is dependent on how NetBackup is configured and how fast the XBSA application sends or receives data. It is important that the developers of an XBSA application have some understanding of NetBackup to get the most out of it. Most of that is determined by any individual implementation. But there are areas where the application can make a difference in performance.

Here are some hints to help you get the most out of the NetBackup XBSA interface.

- Use copyId if you know it. If the XBSA application has the ability to know or keep the copyId for further reference, this is the most efficient method of getting the object for restore.
- Specify dates when doing a query. If start and end dates are not specified when doing a query, the NetBackup XBSA interface may need to search through the entire NetBackup catalog to find the object. Specifying dates allows it to narrow its search.
- Limit use of wildcards when doing a query. Wildcards, including the "ANY" value of the enumeration types, cause more overhead searching and can also cause large portions of the NetBackup catalog to be searched. This is especially true of the pathName. Wildcards can be very useful, but from a performance perspective they are harmful.
- Use OBJECT_STATUS_MOST_RECENT. If the XBSA application knows that a pathName is unique, or that it is searching for only the most recent copy of that object, set the objectStatus of the query descriptor to OBJECT_STATUS_MOST_RECENT. This lets NetBackup stop searching once one copy has been found.
- Unless the images are very large, create multiple objects per transaction rather than one object per transaction when there are multiple objects being created. Every transaction creates a NetBackup job that must be scheduled, open sockets, mount tapes, and so on. For large objects, this overhead is dwarfed by the time it takes to backup the data. On the other hand if there are many small objects, this overhead becomes significant. Note that when creating multiple objects with one transaction, there is no mechanism in the NetBackup XBSA interface to delete a single object. However, the entire NetBackup image can be deleted by calling NBBSADeleteImage().

Chapter 4

How to build an XBSA application

This chapter includes the following topics:

Getting help

Included in the NetBackup DataStore SDK are XBSA sample files that can be used as a basis for creating an application. Included are both source files and Makefiles. See "What the sample files do" on page 136. The Makefiles included in the sample directory can be used as a basis for setting up an environment for creating an XBSA application.

Flags and defines

There are no specific flags or defines that need to be used to compile using the NetBackup XBSA interface. You should be able to use any values to make your application compile efficiently.

How to build in debug mode

There is no compile level debug mode built into the XBSA libraries or header files. The NetBackup Verbose level controls the debug messages.

How to debug the application

Debugging an XBSA application is best done through the log files generated by NetBackup. This assumes that the XBSA application itself compiles and does not have any known run-time errors.

See Logging and NetBackup for more information on this topic. You should also see the Logging section in the NetBackup System Administrator's Guide, Volume I. The NetBackup Verbose level controls the amount of debug messages that are sent to the logs.

One of the advantages of debugging in this way is that it ties in with the NetBackup logging that is going on for the other NetBackup processes. In many cases, it can be a configuration issue that is causing a failure rather than a problem in the NetBackup XBSA interface or the XBSA application.

Static libraries

The example Makefiles have example entries for using static libraries for your XBSA application. These entries include the path to the static archive library, `libxbsa.a`, along with the system libraries that are also required to be included.

For the UNIX platforms, (from `Makefile.unix`) see the following example:

```
#LIBS    = $(XBSA_SDK_DIR)/lib/HP-UX-IA64/HP-UX11.31/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/Debian3.10.0/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/RedHat3.10.0/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/SuSE4.4.73/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-ppc64le/IBMpSeriesRedHat3.10.0/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-ppc64le/IBMpSeriesSuSE4.4.21/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-s390x/IBMzSeriesRedHat3.10.0/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-s390x/IBMzSeriesSuSE4.4.73/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/RS6000/AIX6/libxbsa64.a -ldl -lc
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris10/libxbsa64.a -lintl -lsocket -lnsl
-ldl -ladm
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris_x86_10_64/libxbsa.a -lintl -lsocket
-lnsl -ldl -ladm
```

For the Windows platforms, see the following example:

```
#LIBS = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows\xbsas.lib
```

Dynamic libraries

The example Makefiles have example entries for using dynamic libraries for your XBSA application.

For the UNIX platforms, (from `Makefile.unix`) see the following example:

```
# Use one of these LIBS to bind dynamically

LIBS = -L/usr/openv/lib -lxbsa -lnbclientcST -lnbbasecST
-lnbtlscST -lnbsslST
#LIBS = -L/usr/openv/lib -lxbsa64 -lnbclientcST -lnbbasecST
-lnbtlscST -lnbsslST
```

For the Windows platforms, (from `Makefile.nt`) see the following example:

```
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows\xbsas.lib
```

The dynamic shared object libraries are installed with the NetBackup client on any supported client platform. Similar to the static libraries, there is a `libxbsa.so` or a `libxbsa64.so`. On UNIX platforms, the libraries are installed to `/usr/openv/lib`. On Windows platforms, the libraries are installed on *`install_directory`*`\netbackup\bin`.

End-user configuration

After an XBSA application has been created and installed on a NetBackup client, a NetBackup policy and schedule must be configured through the NetBackup GUI or command line. See "About How to run a NetBackup XBSA application" on page 73.

Chapter 5

How to run a NetBackup XBSA application

This chapter includes the following topics:

About How to run a NetBackup XBSA application

After an XBSA application has been created, it can be used in a NetBackup environment to store and retrieve data. To use an XBSA application, at least one "DataStore" policy with the appropriate schedules needs to be defined. A configuration can have a single policy that includes all clients or there can be many policies, some of which include only one client.

This manual only contains a brief description of configuring a DataStore policy. More information on creating policies and configuring NetBackup can be found in the NetBackup System Administrator's Guide for UNIX, Volume I, or NetBackup System Administrator's Guide for Windows, Volume I.

Creating a NetBackup policy

A NetBackup policy defines the backup criteria for a specific group of one or more clients. These criteria include:

- storage unit and media to use
- backup schedules
- script files to be executed on the clients
- clients to be backed up

Selecting a storage unit

Each policy sends the data to a defined storage unit. The storage units must have already been defined and one needs to be selected for the DataStore policy.

Adding new schedules

Each policy has its own set of schedules. These schedules control initiation of automatic backups and also specify when user operations can be initiated.

An XBSA application requires each policy to have at least an application backup schedule. To satisfy this requirement, an application backup schedule named Default-Application-Backup is automatically created when you configure a new DataStore policy. The backup window for an application backup schedule must encompass the time period during which all of the NetBackup DataStore jobs, scheduled and unscheduled, occur. This is necessary because the application backup schedule starts processes that are required for all of the XBSA application backups, including those started automatically.

If the user wants NetBackup to initiate the XBSA application, an automatic backup schedule is also required. An automatic backup schedule specifies the dates and times when NetBackup automatically starts backups by running the XBSA scripts in the order that they appear in the Files list. If there is more than one client in the DataStore policy, the XBSA scripts are executed on each client.

Adding script files to the files list

Each policy has a Files list. When a DataStore policy is configured, the Files list is a list of script(s) that are executed when the backup is initiated. That script is executed as a user-directed backup. The script should contain any commands that are required to prepare the application for a backup, including setting up an environment, halting processes, and so on, along with calling the XBSA application with the parameters that are required to execute the backup operations.

Adding new clients

Each policy also has a list of NetBackup clients. This list should contain all clients on which the XBSA application is going to run.

Running a NetBackup XBSA application

Once configured, backups and restores can be run either from the XBSA application or through jobs scheduled through NetBackup. NetBackup can run backups and restores indirectly through the XBSA application by executing scripts that contain the XBSA application backup or restore commands.

Backups and restores initiated by NetBackup (through a script)

The XBSA application can be initiated through NetBackup. This lets the XBSA application be treated like the rest of the system environment when creating and scheduling backup windows and other resource considerations. Backup and restore operations through NetBackup are done through scripts. When a DataStore policy is configured, the Files list is a script that is to be executed when the backup or restore is initiated. That script is executed as a user-directed backup. These scripts should contain any commands that are required to prepare the application for a backup or restore, including setting up an environment, halting processes, and so on, along with calling the XBSA application with the parameters that are required to execute the backup or restore operations.

Note: All scripts must be stored and run locally. One recommendation is that scripts should not be world-writable. Scripts are not allowed to be run from network or remote locations. Any script that is created and saved in the NetBackup `db_ext` (UNIX) or `dbext` (Windows) location needs to be protected during a NetBackup uninstall.

For more information about registering authorized locations and scripts, review the knowledge base article:

http://www.veritas.com/docs/000126002

See "Registering authorized locations used by a NetBackup database script-based policy" on page 142.

There is no ability to browse for backups. The Files list is a script to be executed, not a list of objects that can be restored. It is up to these scripts to determine what needs to be backed up or conversely what XBSA objects need to be restored. In this regard, the XBSA application needs to be fairly intelligent or allow options that can be specified to give the script the ability to be intelligent.

Backups and restores from the command line

The NetBackup XBSA application can also be initiated from the command line to run a backup or restore. Commands included in the backup or restore scripts can also be run directly from the command line. The XBSA application connects to NetBackup through the XBSA interface and a NetBackup job is started. For backups, a backup window must be open in the application backup schedule.

Chapter 6

API reference

This chapter includes the following topics:

Error messages

The following table lists the possible return codes for the NetBackup XBSA functions. The descriptions for each individual function lists the valid return codes that are valid for that function.

The return code BSA_RC_SUCCESS is returned on successful completion by all NetBackup XBSA function calls.

Table 6-1 Error Messages for NetBackup XBSA Functions

Value	Return Code Name	Meaning
0x00	BSA_RC_SUCCESS	The function succeeded.
0x03	BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
0x04	BSA_RC_AUTHENTICATION_FAILURE	There was an authentication failure.
0x05	BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.

Table 6-1 Error Messages for NetBackup XBSA Functions *(continued)*

Value	Return Code Name	Meaning
0x06	BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
0x0B	BSA_RC_INVALID_VOTE	The value specified for vote is invalid.
0x0E	NBBSA_RC_MORE_DATA	There are more objects to restore in a multiple object restore operation.
0x0D	NBBSA_RC_FEATURE_NOT_LICENSED	The license for the requested feature is not available.
0x11	BSA_RC_NO_MATCH	No XBSA object matched the specified predicate.
0x12	BSA_RC_NO_MORE_DATA	No more data is available.
0x15	NBBSA_RC_INVALID_PARAMETER	A parameter passed to this function has an invalid value.
0x1A	BSA_RC_OBJECT_NOT_FOUND	There is no copy of the requested XBSA object.
0x20	BSA_RC_TRANSACTION_ABORTED	The transaction was aborted.
0x34	BSA_RC_INVALID_DATABLOCK	The BSA_DataBlock32 parameter contained an inconsistent value.
0x4B	BSA_RC_VERSION_NOT_SUPPORTED	The NetBackup implementation does not support the specified version of the interface.
0x4D	BSA_RC_ACCESS_FAILURE	Access to the requested XBSA object is not possible.
0x4E	BSA_RC_BUFFER_TOO_SMALL	The supplied buffer is too small to contain the data, as specified by the accompanying size parameter.
0x4F	BSA_RC_INVALID_COPYID	The copyId field contained an unrecognized value.

Table 6-1 Error Messages for NetBackup XBSA Functions *(continued)*

Value	Return Code Name	Meaning
0x50	BSA_RC_INVALID_ENV	An entry in the environment structure is invalid or missing.
0x51	BSA_RC_INVALID_OBJECTDESCRIPTOR	The BSA_ObjectDescriptor was invalid.
0x53	BSA_RC_INVALID_QUERYDESCRIPTOR	The BSA_QueryDescriptor was invalid.
0x55	BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.

Function calls

This section contains the C language definitions for the NetBackup XBSA API functions. The NetBackup XBSA interface includes functions defined by the XBSA specifications and some NetBackup extended functions. Both of these function sets use the type definitions and data structures defined in the next chapter.

The following table lists the XBSA function specifications defined in this topic.

Table 6-2 XBSA Function Specifications

Function Call	Operation
BSAInit	Initialize the environment and set up a session
BSATerminate	Terminate a session
BSABeginTxn	Begin a transaction
BSAEndTxn	End a transaction
BSACreateObject	Create an XBSA object
BSASendData	Send a byte stream of data in a buffer
BSAGetObject	Get an XBSA object
BSAGetData	Get a byte stream of data using buffers
BSAEndData	End a BSAGetData() or BSASendData() sequence
BSADeleteObject	Delete an XBSA object

Table 6-2 XBSA Function Specifications *(continued)*

Function Call	Operation
BSAQueryObject	Query about XBSA object copies
BSAGetNextQueryObject	Get the next XBSA object relating to a previous query
BSAGetEnvironment	Retrieve the current environment for the session
BSAGetLastError	Retrieve the error code for the last system error
BSAQueryApiVersion	Query for the current version of the API
BSAQueryServiceProvider	Query the name of NetBackup implementation

The following table lists the NetBackup XBSA function extensions defined later in this topic. These functions are provided by the NetBackup XBSA interface to enhance the usability and performance of an XBSA application used with NetBackup. The use of these functions is not required. An application using strictly XBSA functions is supported.

Table 6-3 NetBackup XBSA Function Extensions

Function Call	Operation
NBBSAAddToMultiObjectRestoreList	Add objects to a list of objects to be restored in one job
NBBSADeleteImage	Delete a NetBackup image
NBBSAEndGetMultipleObjects	End the restore of multiple objects
NBBSAFreeJobInfo	Free job information
NBBSAGetEnv	Get the value of a single XBSA environment value
NBBSAGetErrorString	Get the string error message of an XBSA error code
NBBSAGetJobId	Get the job ID for a backup or restore transaction
NBBSAGetJobInfo	Get the job information
NBBSAGetMediaIds	Get the media IDs for a NetBackup image
NBBSAGetMultipleObjects	Initiate a restore of a list of objects
NBBSAGetServerError	Get the NetBackup error code and text from the NetBackup server
NBBSALogMsg	Log a message to the XBSA logs

Table 6-3 NetBackup XBSA Function Extensions *(continued)*

Function Call	Operation
NBBSASetEnv	Set the value of a single XBSA environment value
NBBSAUpdateEnv	Update the current environment for the session
NBBSAValidateFeatureId	Validate the license key for the specified feature ID

Conventions

The following conventions are used to indicate input or output for parameters:

(I) indicates input

(O) indicates output

(I/O) indicates input and output

In many cases, the actual input parameter is a pointer to a data structure. In these cases the terms "I", "O" and "I/O" refer to changes in the value of the data structure rather than to changes in the value of the pointer itself.

Function specifications

The following is a list of the function specifications for XBSA.

BSABeginTxn

Begin a transaction.

SYNOPSIS

#include <xbsa.h>

int BSABeginTxn(BSA_Handle bsaHandle)

DESCRIPTION

The BSABeginTxn() call indicates to the NetBackup XBSA interface the beginning of one or more actions that are executed as an atomic unit, that is, all of the actions will succeed or none will succeed. An action can be assumed to be either a single function call or a series of function calls that are made for a particular purpose.

For example, a BSACreateObject() call followed by a number of BSASendData() calls and terminated by a BSAEndData() call can be considered to be a single action.

In normal use, a BSABeginTxn() call is always coupled with a subsequent BSAEndTxn() call. If BSATerminate() is called during a transaction, the transaction is aborted.

If BSA_SERVICE_HOST has not been specified before calling BSABeginTxn(), the default NetBackup server is determined and the feature is checked for a valid license key. The feature_ID is the default DataStore feature_ID unless a specific NetBackup feature_ID has been specified using NBBSA_FEATURE_ID. If a valid license is not found, the transaction returns a NBBSA_RC_FEATURE_NOT_LICENSED error and not open the transaction.

Nested transactions are not supported.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect. Nested transactions are not supported.
NBBSA_RC_FEATURE_NOT_LICENSED	The license for the requested feature is not available.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_SUCCESS	The function succeeded.

BSACreateObject

Create an XBSA object.

SYNOPSIS

#include <xbsa.h>

int BSACreateObject(BSA_Handle bsaHandle, BSA_ObjectDescriptor *objectDescriptorPtr, BSA_DataBlock32 *dataBlockPtr)

DESCRIPTION

The BSACreateObject() call creates an XBSA object within NetBackup. Duplicate BSA_ObjectNames are allowed.

The BSACreateObject() call is used to create an XBSA object based on the information in the objectDescriptor. This call initiates the communication between the NetBackup XBSA interface and the NetBackup server. The XBSA object data can then be passed in memory buffers. The dataBlockPtr parameter in the BSACreateObject() call allows the caller to obtain information about the buffer structure required by the NetBackup XBSA interface.

The XBSA object data is passed through one or more BSASendData() calls. If there is no data to be sent, then a BSAEndData() call must be used to indicate completion of the XBSA object. The BSASendData() and BSAEndData() calls must follow the BSACreateObject() call and must be in the same transaction.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_ObjectDescriptor *objectDescriptorPtr (I/O)	This parameter is used to pass XBSA object attributes, including its name, copy type, and so on.
BSA_DataBlock32 *dataBlockPtr (O)	This parameter is a pointer to a structure that is used to obtain the details of the required buffer structure.

EXTENDED DESCRIPTION

Within the XBSA object descriptor, all fields must contain valid values. Enumerations must contain one of their enumerated values. Strings must be null-terminated. All other fields must be in the range of valid values for that field.

The following fields in the XBSA object descriptor are optional: objectOwner, objectDescription, and objectInfo. The optional value for either field of objectOwner and the field objectDescription is the empty string. The optional value for objectInfo is all zeros. If the bsa_ObjectOwner is empty, it defaults to the value specified in BSAInit().

Note: For NetBackup XBSA Version 1.1.0, the NetBackup XBSA interface and NetBackup determine the XBSA object ownership. If the bsa_ObjectOwner field is specified, it is saved with the object but does not define ownership.

The following fields in the XBSA object descriptor are mandatory: objectName, copyType, estimatedSize, resourceType, and objectType. For objectName this

means that the pathName must contain a non-empty string. For copyType and objectType the enumeration value "ANY" is not allowed.

The estimatedSize must contain a non-zero estimate if the XBSA application intends to create a non-empty XBSA object (that is, there is XBSA object data). This size is in bytes. If the estimatedSize is zero, this call must be followed by a BSAEndData() without calling BSASendData() in between. There are no resource allocations based on this estimate, only whether the object will have data or not, so the estimate does not need to be accurate.

The NetBackup XBSA interface can return several values to the XBSA application through the objectDescriptorPtr for a newly created XBSA object. The interface returns either all or none of these values.

The copyId attribute is a persistent, fixed-length object identifier that remains unchanged throughout the life of the XBSA object.

Note: For NetBackup XBSA Version 1.1.0, the copyId is only guaranteed to be unique on a given NetBackup master server.

If the copyId field is non-zero, the NetBackup XBSA interface returned values for the copyId, createTime, restoreOrder, and objectStatus fields.

The createTime field is in UTC. The restoreOrder field can have the value zero, which means that the NetBackup XBSA interface did not specify a restore order.

The dataBlockPtr structure does not point to an actual data buffer. All of the values in the dataBlockPtr should be zero, and they are overwritten. The structure is used by the NetBackup XBSA interface to provide the XBSA application with the interface preference for the structure of the data blocks that are used to pass the NetBackup XBSA object's data. The XBSA application should examine the values returned to determine the buffer structure that it should create. The significance of the returned values is as follows:

bufferLen == 0	NetBackup has no restrictions on the buffer length. No trailer portion is required.
bufferLen != 0	NetBackup accepts buffers that are at least bufferLen bytes in length (minimum length). The length of the trailer portion of buffers is: trailerBytes >= (bufferLen - numBytes - headerBytes)
numBytes == 0	NetBackup has no restrictions on the length of the data portion of the buffer.
numBytes != 0	The maximum length of the data portion of buffers accepted by NetBackup must not exceed numBytes bytes.

headerBytes == 0	NetBackup only accepts buffers with no header portion.
headerBytes != 0	The length of the header portion of buffers accepted by NetBackup is headerBytes bytes.
bufferPtr	Not used

The values returned by the call to BSACreateObject() remain in effect for the duration of the data transfer of the XBSA object being created, that is, until the next BSAEndData() call. The NetBackup XBSA interface currently does not have any header or trailer requirements, so the full buffer specified can be used by the XBSA application. This is documented for completeness with the XBSA specification and to allow for future use of these fields as specified by the XBSA specifications.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Cannot create the XBSA object with the given descriptor.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_DATABLOCK	The BSA_DataBlock32 parameter contained an inconsistent value.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_INVALID_OBJECTDESCRIPTOR	The BSA_ObjectDescriptor was invalid.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments
BSA_RC_SUCCESS	The function succeeded.

BSADeleteObject

Delete a NetBackup XBSA object.

SYNOPSIS

#include <xbsa.h>

int BSADeleteObject(BSA_Handle bsaHandle, BSA_UInt64 copyId)

DESCRIPTION

The BSADeleteObject() call only deletes an XBSA object from NetBackup. The value for copyId can be obtained from a previous BSAQueryObject() call. The copyId value is unique on a given NetBackup master server. An XBSA application can only delete the NetBackup XBSA objects that it owns.

BSADeleteObject() only works when there is only one object in an image, that is, one object created per transaction. If there are multiple objects, BSADeleteObject() returns a BSA_RC_SUCCESS status, but the object still exists.

The actual delete of the object from NetBackup occurs when the transaction is closed with a commit. A query in the same transaction can still return the object. If the transaction is aborted, the object is not deleted.

If the object data is stored in a NetBackup disk storage unit, the data is deleted with the object. If the object is on a tape storage unit, the data is considered expired but is not deleted until all of the objects on the media are expired. You can create and then delete the same NetBackup XBSA object within a single transaction.

Note: Objects which are found, but are protected in the NetBackup catalog (Legal Hold, WORM storage, etc.) are not deleted and a `BSA_RC_SUCCESS` status is returned on the `BSAEndTxn` call.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_UInt64 copyId (I)	This parameter is the unique ID of the XBSA object to be deleted. The value(s) for a specific XBSA object can be obtained through a BSAQueryObject() call.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Cannot delete an XBSA object with the given copyId.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_COPYID	The copyId field cannot be zero.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.

BSA_RC_OBJECT_NOT_FOUND	The given copyId does not exist.
BSA_RC_SUCCESS	The function succeeded.

BSAEndData

End a BSAGetData() or BSASendData() sequence.

SYNOPSIS

#include <xbsa.h>

int BSAEndData(BSA_Handle bsaHandle)

DESCRIPTION

The caller issues BSAEndData() after a call to BSACreateObject() followed by zero or more BSASendData() calls, or after a call to BSAGetObject() followed by zero or more BSAGetData() calls to signify the end of data. When used with BSAGetObject() or BSAGetData() calls, BSAEndData() does not transfer any more data for the NetBackup XBSA object to the caller. When used with BSACreateObject() or BSASendData() calls, BSAEndData() tells the NetBackup XBSA interface that the caller has finished sending data for a particular NetBackup XBSA object. BSAEndData() signifies the end of data for the immediately preceding BSACreateObject(), BSAGetObject(), BSAGetData(), or BSASendData().

It is also required after a call to BSAGetObject() or BSACreateObject() if the object is empty.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_SUCCESS	The function succeeded.

BSAEndTxn

End a transaction.

SYNOPSIS

#include <xbsa.h>

int BSAEndTxn(BSA_Handle bsaHandle, BSA_Vote vote)

DESCRIPTION

BSAEndTxn() is coupled with BSABeginTxn() to identify the API call or set of API calls that are to be treated as a transaction. The caller must specify as a parameter to the BSAEndTxn() call whether or not the transaction is to be committed.

The BSA_RC_TRANSACTION_ABORTED error is only returned when a vote of BSA_Vote_COMMIT has been specified but an error has occurred that causes the transaction to be aborted. A BSAEndTxn() specifying a vote of BSA_Vote_ABORT returns a success status.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_Vote vote (I)	This parameter indicates whether or not the caller wants to commit all the actions done between the previous BSABeginTxn() call and this call.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	There is no corresponding BSABeginTxn().
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_INVALID_VOTE	The value specified for vote is invalid.
BSA_RC_SUCCESS	The function succeeded.
BSA_RC_TRANSACTION_ABORTED	The transaction was aborted.

BSAGetData

Get a byte stream of data using buffers.

SYNOPSIS

#include <xbsa.h>

int BSAGetData(BSA_Handle bsaHandle, BSA_DataBlock32 *dataBlockPtr)

DESCRIPTION

BSAGetData() allows the caller to request a buffer full of XBSA object data from the NetBackup XBSA interface. This call is used after a BSAGetObject() call or after other BSAGetData() calls.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_DataBlock32 *dataBlockPtr (I/O)	This parameter is a pointer to a structure that includes both a pointer to the buffer for the data that is to be received and the size of the buffer. Further, the API returns, in this structure, the number of bytes of data that have been sent to the caller for this call.

EXTENDED DESCRIPTION

The NetBackup XBSA interface overwrites the numBytes field to provide the actual values used. The NetBackup XBSA interface does not modify any other fields. The XBSA application can only use the data portion of the buffer in which the XBSA object data is contained.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_DATABLOCK	The BSA_DataBlock32 parameter contained an inconsistent value.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_NO_MORE_DATA	There is no more data.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_SUCCESS	The function succeeded.

BSAGetEnvironment

Retrieve the current NetBackup XBSA environment variables for the session.

SYNOPSIS

#include <xbsa.h>

int BSAGetEnvironment(BSA_Handle bsaHandle, BSA_UInt32 *sizePtr, char **environmentPtr)

DESCRIPTION

The BSAGetEnvironment() call returns the (keyword, value) pairs that are currently defined in the NetBackup XBSA environment for the session

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_UInt32 *sizePtr (I/O)	This parameter contains the size of the environment buffer in bytes.
char **environmentPtr (O)	This parameter is a pointer to an array of character pointers to the environment variables strings for the session. Each string consists of a keyword followed by an 10.0=10.0 followed by a null-terminated value. A NULL pointer terminates the array of pointers.

EXTENDED DESCRIPTION

If a buffer too small error is encountered, the required size is returned in the sizePtr parameter. If the sizePtr parameter is set to zero, it forces a buffer too small error, that provides a mechanism to query the required size.

See “Environment variable definitions” on page 25.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_BUFFER_TOO_SMALL	The size of the data buffer is invalid.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.

BSA_RC_SUCCESS	The function succeeded.

BSAGetLastError

Return the last system error code.

SYNOPSIS

#include <xbsa.h>

int BSAGetLastError(BSA_UInt32 *sizePtr, char *errorCodePtr)

DESCRIPTION

The BSAGetLastError() call returns a textual description of the last error encountered by the NetBackup XBSA interface. It is used to return the NetBackup-specific information that describes the underlying cause of the failure of the most recent XBSA call; for example, a network failure.

PARAMETERS

BSA_UInt32 sizePtr (I/O)	This parameter contains the size of the error buffer in bytes.
char *errorPtr (O)	This parameter is a pointer to a data area that contains a text string describing the last error encountered.

EXTENDED DESCRIPTION

If the NetBackup XBSA interface sets the sizePtr parameter to zero, it is unable to return a string describing the last error. This indicates that the NetBackup XBSA interface has no record of what error occurred.

If a BSA_RC_BUFFER_TOO_SMALL error is encountered, the required size is returned in the sizePtr parameter. If the XBSA application sets the sizePtr parameter to zero, this forces a BSA_RC_BUFFER_TOO_SMALL error that provides a mechanism to query the required size.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_BUFFER_TOO_SMALL	The size of the data buffer is invalid.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.

BSA_RC_SUCCESS	The function succeeded.

BSAGetNextQueryObject

Get the next NetBackup XBSA object found from a previous query.

SYNOPSIS

#include <xbsa.h>

int BSAGetNextQueryObject(BSA_Handle bsaHandle, BSA_ObjectDescriptor *objectDescriptorPtr)

DESCRIPTION

The BSAGetNextQueryObject() call returns the next NetBackup XBSA object descriptor that is a member of a previous query. Successive calls to BSAGetNextQueryObject() return all of the NetBackup XBSA object descriptors from a query one object at a time. When the last object descriptor from a query has been found, the function returns a status of BSA_RC_NO_MORE_DATA.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_ObjectDescriptor *objectDescriptorPtr (O)	This parameter is a pointer to an XBSA object descriptor structure that is populated with the values from the next XBSA object in the list generated by the query.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_NO_MORE_DATA	There is no more data.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments
BSA_RC_SUCCESS	The function succeeded.

BSAGetObject

Get an object.

SYNOPSIS

#include <xbsa.h>

int BSAGetObject(BSA_Handle bsaHandle, BSA_ObjectDescriptor *objectDescriptorPtr, BSA_DataBlock32 *dataBlockPtr)

DESCRIPTION

BSAGetObject() retrieves the NetBackup XBSA object identified by the copyId and prepares the NetBackup XBSA Interface to retrieve the XBSA object data. It initiates the communication with the NetBackup server to retrieve the object.

The dataBlockPtr parameter in the BSAGetObject() call allows the caller to obtain information about the buffer structure required by the NetBackup XBSA interface. The caller obtains the NetBackup XBSA object data through subsequent BSAGetData() calls. The caller must terminate the receipt of the data by using the BSAEndData() call.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_ObjectDescriptor *objectDescriptorPtr (I)	This parameter is a pointer to a data area used to pass the NetBackup XBSA object's copyId to the NetBackup XBSA interface.
BSA_DataBlock32 *dataBlockPtr (O)	This parameter is a pointer to a structure that is used to obtain the details of the required buffer structure.

EXTENDED DESCRIPTION

It is mandatory that the copyId field in the BSA_ObjectDescriptor structure is set as this is the only field that is checked. A copyId value of zero cannot identify a valid XBSA object. BSAGetObject() matches the copyId field for equality.

The dataBlockPtr structure does not point to an actual buffer. All values in the dataBlockPtr should be zero, and are overwritten. The structure is used by the NetBackup XBSA interface to provide the XBSA application with the interface's preference for the structure of the data blocks that are used to pass the NetBackup XBSA object's data. The XBSA application should examine the values returned to determine the buffer structure that it should create. The significance of the returned values is as follows:

bufferLen == 0	NetBackup has no restrictions on the buffer length. No trailer portion is required.
bufferLen != 0	NetBackup accepts buffers that are at least bufferLen bytes in length (minimum length). The length of the trailer portion of buffers is: trailerBytes >= (bufferLen - numBytes - headerBytes).
numBytes == 0	NetBackup has no restrictions on the length of the data portion of the buffer.
numBytes != 0	The minimum length of the data portion of buffers accepted by NetBackup must be numBytes bytes. If the interface provides a larger data portion, NetBackup may take advantage of it.
headerBytes == 0	NetBackup only accepts buffers with no header portion.
headerBytes != 0	The length of the header portion of buffers accepted by NetBackup is headerBytes bytes.
bufferPtr	Not used.

The values returned by the call to BSAGetObject() remain in effect for the duration of the data transfer of the NetBackup XBSA object being retrieved, that is, until the next BSAEndData() call. The NetBackup XBSA interface currently does not have any header or trailer requirements, so the full buffer specified can be used by the XBSA application. This is documented for completeness with the XBSA specification and to allow future use of these fields as specified by the XBSA specifications.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Access to the requested XBSA object is not possible. Cannot retrieve the XBSA object with the given copyId.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_COPYID	The copyId cannot be zero.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.

BSA_RC_OBJECT_NOT_FOUND	The given copyId does not exist.
BSA_RC_SUCCESS	The function succeeded.

BSAInit

Initialize the environment and set up a session.

SYNOPSIS

#include <xbsa.h>

int BSAInit(BSA_Handle *bsaHandlePtr, BSA_SecurityToken *tokenPtr, BSA_ObjectOwner *objectOwnerPtr,

char **environmentPtr)

DESCRIPTION

The BSAInit() call authenticates the XBSA application, sets up a session with the NetBackup XBSA interface and an environment for subsequent API calls for the caller. Nested sessions are not supported.

PARAMETERS

BSA_Handle *bsaHandlePtr (O)	This parameter is used to return the handle that identifies this session and must be used for subsequent API calls using this session.
BSA_SecurityToken *tokenPtr (I)	For NetBackup XBSA Version 1.1.0, this parameter is ignored. Client authentication is part of NetBackup functionality and is performed between the NetBackup XBSA interface and the NetBackup server.
BSA_ObjectOwner *objectOwnerPtr (I)	This parameter is a pointer to a structure used to specify both the bsa_ObjectOwner and the app_ObjectOwner. For NetBackup XBSA Version 1.1.0, the NetBackup XBSA interface and NetBackup determine object ownership. If the bsa_ObjectOwner field is specified, it is ignored. The app_ObjectOwner is optional and can be the empty string. The BSA_ObjectOwner established when the session is created is used in subsequent authorization checking.

char **environmentPtr (I)	This parameter is a pointer to a structure that contains the new NetBackup XBSA environment variables (keyword, value) pairs, for the session. The new NetBackup XBSA environment consists of a pointer to an array of strings. Each string consists of a keyword followed by an '=' and followed by a null-terminated value. No spaces are allowed around the '='. A NULL pointer terminates the array of pointers.

EXTENDED DESCRIPTION

See "Environment variable definitions" on page 25.

Variables defined by the XBSA application but not interpreted by the NetBackup XBSA interface are silently ignored and not added to the NetBackup XBSA environment. Variables required by the NetBackup XBSA interface and not specified by the application can result in a BSA_RC_INVALID_ENV error during a BSAInit() call. The BSAGetEnvironment() call only returns the NetBackup XBSA environment variables that are meaningful to the NetBackup XBSA interface. This allows the XBSA application to discover which variables specified in the call to BSAInit() the NetBackup XBSA interface interpreted.

When an XBSA application connects to a NetBackup XBSA interface, it can make an initial call to BSAQueryApiVersion() to determine the highest version of the specification supported. If the application supports that version, it should specify it when calling BSAInit(). If the application does not support that version, or did not call BSAQueryApiVersion(), the XBSA application should specify the version it requires. If a "version not supported" error is encountered, and the XBSA application supports other versions, the XBSA application may retry the call to BSAInit() specifying a different version.

BSAInit() can also determine the verbose level and open the log file if the log directory exists. Thus the XBSA application can start logging after BSAInit().

If BSA_SERVICE_HOST and NBBSA_FEATURE_ID are specified in the list of XBSA environment variables, the feature is checked for a valid license key. If a valid license is not found, the transaction returns a NBBSA_RC_FEATURE_NOT_LICENSED error and does not open the session.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_AUTHENTICATION_FAILURE	There was an authentication failure.

BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect. Nested sessions are not supported.
BSA_RC_INVALID_ENV	An entry in the environment structure is invalid or missing.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
NBBSA_RC_FEATURE_NOT_LICENSED	The license for the requested feature is not available.
BSA_RC_SUCCESS	The function succeeded.
BSA_RC_VERSION_NOT_SUPPORTED	The NetBackup XBSA interface does not support the specified version of the interface.

BSAQueryApiVersion

Query for the current version of the API.

SYNOPSIS

#include <xbsa.h>

int BSAQueryApiVersion(BSA_ApiVersion *apiVersionPtr)

DESCRIPTION

The BSAQueryApiVersion() call is used to determine the current version of the NetBackup XBSA interface. The version information consists of the issue, version within the issue, and level within the version. If the NetBackup XBSA interface supports more than one version, the latest version information is returned.

PARAMETERS

BSA_ApiVersion *apiVersionPtr (O)	This parameter is a pointer to a structure that is to be used to return the current issue, version, and level, of the API.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_NULL_ARGUMENT	A NULL apiVersionPtr was encountered.
BSA_RC_SUCCESS	The function succeeded.

BSAQueryObject

Query about the XBSA object copies.

SYNOPSIS

#include <xbsa.h>

int BSAQueryObject(BSA_Handle bsaHandle, BSA_QueryDescriptor *queryDescriptorPtr,

BSA_ObjectDescriptor *objectDescriptorPtr)

DESCRIPTION

The BSAQueryObject() call initiates a request for information on NetBackup XBSA object copies from the NetBackup XBSA interface. The results of the query are determined by the search conditions specified in the query descriptor. The XBSA object descriptor for the first XBSA object satisfying the query search conditions is returned in the BSA_ObjectDescriptor (referenced by the objectDescriptorPtr parameter). The application can obtain the other XBSA object descriptors found by the query by successive calls to BSAGetNextQueryObject().

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_QueryDescriptor *queryDescriptorPtr (I)	This parameter is a pointer to a structure that contains the search conditions for the query.
BSA_ObjectDescriptor *objectDescriptorPtr (O)	This parameter is a pointer to a structure that is used to return the XBSA object descriptor for the first XBSA object that satisfies the search condition specified in the query.

EXTENDED DESCRIPTION

This function may only be used as part of a retrieval transaction.

A limited wildcard capability is available as follows:

Data Type	Wildcard options
string	"*" matches 0 or more characters "?" matches exactly one character "*" matches a literal "*" "\?" matches a literal "?" "\\" matches a literal "\" String matching is performed without any interpretation of the string contents. There is no implied knowledge of the structure of the string contents.

time	Zero value = any time
enumerations	ANY value matches any value
BSA_ObjectOwner	Defaults to value specified at session initialization

The following examples illustrate wildcard string matching:

BSA_ObjectName.pathName = /server/*	Matches all NetBackup XBSA objects for this server
BSA_ObjectName.pathName = /server/rootdbs/*	Matches all levels of rootdbs
BSA_ObjectName.pathName = /server/????	Matches all levels whose name is exactly four characters long

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Access to the requested NetBackup XBSA object descriptor is not permitted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_INVALID_QUERYDESCRIPTOR	The BSA_QueryDescriptor was invalid.
BSA_RC_NO_MATCH	No NetBackup XBSA objects matched the given query.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_SUCCESS	The function succeeded.

BSAQueryServiceProvider

Retrieve a string identifying NetBackup provider.

SYNOPSIS

#include <xbsa.h>

int BSAQueryServiceProvider(BSA_UInt32 *sizePtr, char *delimiter, char *providerPtr)

DESCRIPTION

The BSAQueryServiceProvider() call returns a hierarchical string identifying NetBackup provider.

PARAMETERS

BSA_UInt32 *sizePtr (I/O)	This parameter contains the size of the provider buffer in bytes.
char *delimiter (O)	This parameter is a pointer to the character that is used to delimit fields in the provider hierarchical string.
char *providerPtr (O)	This parameter is a pointer to a data area that contains hierarchical string which conveys information identifying NetBackup provider.

EXTENDED DESCRIPTION

The format of the provider string is the same as the NetBackup XBSA environment variable BSA_SERVICE_PROVIDER (see BSAGetEnvironment). The delimiter character is returned in the delimiter parameter.

If a BSA_RC_BUFFER_TOO_SMALL error is encountered, the required size is returned in the sizePtr parameter. If the XBSA application sets the sizePtr parameter to zero, it forces a BSA_RC_BUFFER_TOO_SMALL error that provides a mechanism to query the required size.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_BUFFER_TOO_SMALL	The size of the data buffer is invalid.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_SUCCESS	The function succeeded.

BSASendData

Send a byte stream of data in a buffer.

SYNOPSIS

#include <xbsa.h>

int BSASendData(BSA_Handle bsaHandle, BSA_DataBlock32 *dataBlockPtr)

DESCRIPTION

BSASendData() sends a byte stream of data to the NetBackup XBSA interface in a buffer. BSASendData() can be called multiple times, in case the byte stream of data to be sent is large. This call can be used only after a BSACreateObject() or another BSASendData() call.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.
BSA_DataBlock32 *dataBlockPtr (I)	This parameter is a pointer to a structure that includes a pointer to the buffer from which the data is to be sent, as well as the size of the buffer.

EXTENDED DESCRIPTION

The NetBackup XBSA interface does not overwrite any of the fields in the BSA_DataBlock32 structure. The NetBackup XBSA interface can write into the header and trailer portions of the buffer. See "Use of BSA_DataBlock32 in BSASendData()" on page 23.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_DATABLOCK	The BSA_DataBlock32 parameter contained an inconsistent value.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_SUCCESS	The function succeeded.

BSATerminate

Terminate a session.

SYNOPSIS

#include <xbsa.h>

int BSATerminate(BSA_Handle bsaHandle)

DESCRIPTION

The BSATerminate() call terminates the session with the NetBackup XBSA interface that was set up by a previous BSAInit() call and is associated with the bsaHandle. It also releases any resources acquired for the session, including closing any log files. If BSATerminate() is called within a transaction, the transaction is aborted.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_SUCCESS	The function succeeded.

NBBSAAddToMultiObjectRestoreList

Add objects to a list of objects to be restored in one job.

SYNOPSIS

#include <nbbsa.h>

int NBBSAAddToMultiObjectRestoreList(BSA_Handle bsaHandle, NBBSA_DESCRIPT_LIST ** DescriptList, BSA_ObjectDescriptor * ObjectDescriptorPtr)

DESCRIPTION

NBBSAAddToMultiObjectRestoreList() adds the objectDescriptor passed in to a linked list of objectDescriptors. This list is used when restoring multiple objects. The memory allocated in this function for the list is freed in NBBSAEndGetMultipleObjects().

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
NBBSA_DESCRIPT_LIST **DescriptList (I)	The address of a pointer to a list of BSA_ObjectDescriptor's.
BSA_ObjectDescriptor *ObjectDescriptorPtr (I)	Pointer to an BSA_ObjectDescriptor to be added to the list.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_SUCCESS	The object descriptor has been added to the list.

NBBSADeleteImage

Delete a NetBackup image.

SYNOPSIS

#include <nbbsa.h>

int NBBSADeleteImage(BSA_Handle bsaHandle, BSA_UInt64 copyId)

DESCRIPTION

NBBSADeleteImage() deletes a NetBackup image given the copyId of an XBSA object. The value for copyId can be obtained from a previous BSAQueryObject() call. The copyId value is unique on a given NetBackup Master Server. The XBSA application can only delete a NetBackup image that contains the XBSA objects that it owns.

NBBSADeleteImage() deletes an image regardless of the number of objects contained in an image. For an image containing multiple objects, NBBSADeleteImage() only needs to be called once passing in the copyId of any of the objects contained in the image.

The deletion of an image occurs during the NBBSADeleteImage() processing.

If the image data is stored in a NetBackup disk storage unit, the data is deleted with the image. If the image is on a tape storage unit, the data is considered expired but is not deleted until all of the images on the media are expired. You cannot create and then delete an image within a single transaction.

PARAMETERS

BSA_Handle bsaHandle (I)	This parameter is the handle that associates this call with a previous BSAInit() call.

BSA_UInt64 copyId (I)	This parameter is the unique ID of an XBSA object that is contained in the image to be deleted. The value for a specific XBSA object can be obtained through a BSAQueryObject() call.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Cannot delete NetBackup image with the given copyId.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect. Within a transaction calling NBBSADeleteImage(), it is permissible to embed BSAQueryObject() and BSAGetNextQueryObject() calls. This allows the XBSA application to intermix NBBSADeleteImage() calls with BSAQueryObject() and BSAGetNextQueryObject() calls to delete multiple images within one transaction. Backup and restore operations are not allowed within a transaction that calls NBBSADeleteImage().
BSA_RC_INVALID_COPYID	The copyId field cannot be zero.
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_OBJECT_NOT_FOUND	The given copyId does not exist.
BSA_RC_SUCCESS	The function succeeded.

NBBSAEndGetMultipleObjects

End the restore of multiple objects.

SYNOPSIS

#include <nbbsa.h>

int NBBSAEndGetMultipleObjects(BSA_Handle bsaHandle, BSA_Vote vote, NBBSA_DESCRIPT_LIST * descriptList)

DESCRIPTION

NBBSAEndGetMultipleObjects() closes the communications to the NetBackup server to end a multiple object restore. The objectDescriptor list is freed and a check is made to see if all of the objects requested were restored. If all of the objects were

not restored, NBBSAEndGetMultipleObjects() returns an error. A vote parameter is provided to allow the multiple object restore to be aborted. As with a single object restore, commit or abort provides no functional difference to NetBackup.

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
BSA_Vote vote (I)	Allows the multiple object restore to be committed or aborted.
NBBSA_DESCRIPT_LIST * descriptList (I)	List of objects that were restored as part of the multiple object restore.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit()() call is invalid.
BSA_RC_MORE_DATA	Not all of the requested objects were restored.
BSA_RC_SUCCESS	The end of the restore has been successfully completed.

NBBSAFreeJobInfo

Free job information.

SYNOPSIS

#include <nbbsa.h>

int NBBSAFreeJobInfo(char **jobInfo)

DESCRIPTION

NBBSAFreeJobInfo() frees the job information storage that was allocated previously through a call to NBBSAGetJobInfo().

PARAMETER

char **JobInfo(I)	The job information array of pointers to character strings that was populated by a previous call to NBBSAGetJobInfo().

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered by the argument.
BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_SUCCESS	The function succeeded.

NBBSAGetEnv

Set the value of a single XBSA environment value.

SYNOPSIS

#include <nbbsa.h>

int NBBSAGetEnv(BSA_Handle bsaHandle, char *EnvVar, char *EnvVal, int *ValSize)

DESCRIPTION

NBBSAGetEnv() gives the XBSA application the ability to retrieve the value of a specific XBSA environment variable. The same results can be achieved by calling BSAGetEnvironment() and evaluating for the specific variable being sought.

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
char *EnvVar (I)	Pointer to a null-terminated string that specifies the environment variable.
char *EnvVal (O)	Pointer to a buffer to receive the value of the specified environment variable.
int *ValSize (I/O)	Pointer to the size, in characters, of the buffer pointed to by the EnvVal parameter. Returns the size of EnvVal.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_BUFFER_TOO_SMALL	The buffer pointed to by EnvVal is not large enough, The buffer size, in characters, required to hold the value string and its terminating null character is stored in the location pointed to by ValSize.

BSA_RC_INVALID_ENV	The specified environment variable name was not found in the XBSA environment block for the current session.
BSA_RC_SUCCESS	The function succeeded.

NBBSAGetErrorString

Get the textual error message of an XBSA error code.

SYNOPSIS

#include <nbbsa.h>

int NBBSAGetErrorString(int ErrCode, BSA_UInt32 *sizePtr, char *errorCodePtr)

DESCRIPTION

The NBBSAGetErrorString() call returns a textual description of the XBSA error code passed in.

PARAMETERS

int ErrCode (I)	The XBSA error code.
BSA_UInt32 *sizePtr (I/O)	Pointer to the size, in characters, of the buffer pointed to by the errorCodePtr parameter. Returns the size of errorCodePtr.
char *errorCodePtr (O)	Pointer to a buffer to receive the text of the error code.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_BUFFER_TOO_SMALL	The size of the data buffer is invalid.
BSA_RC_NO_MATCH	No description for error code passed in.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_SUCCESS	The function succeeded.

NBBSAGetJobId

Obtain the job ID for a backup or restore transaction.

SYNOPSIS

#include <nbbsa.h>

int NBBSAGetJobId(BSA_Handle bsaHandle, int *jobId)

DESCRIPTION

NBBSAGetJobId() returns the job ID associated with a backup or restore transaction. To retrieve the job ID for a backup transaction, NBBSAGetJobId() can be called any time after the BSACreateObject() call, but before the BSAEndTxn() call. To retrieve the job ID for a restore transaction, NBBSAGetJobId() can be called any time after the BSAGetObject() call, but before the BSAEndTxn() call. To retrieve the job ID for a multiple object restore transaction, NBBSAGetJobId() can be called any time after the NBBSAGetMultipleObjects() call, but before the BSAEndTxn() call.

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
int *jobId (O)	Pointer to the job ID for the current backup or restore.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect. This code is returned if the current transaction is not a backup or restore transaction. During a backup transaction, this code is returned if NBBSAGetJobId() is called before BSACreateObject() or after BSAEndTxn(). During a restore transaction, this code is returned if NBBSAGetJobId() is called before BSAGetObject () or after BSAEndTxn(). During a multiple object restore transaction, this code is returned if NBBSAGetJobId() is called before NBBSAGetMultipleObjects() or after BSAEndTxn().
BSA_RC_INVALID_HANDLE	The handle used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_SUCCESS	The function succeeded.

NBBSAGetJobInfo

Obtain job information.

SYNOPSIS

#include <nbbsa.h>

int NBBSAGetJobInfo(BSA_Handle bsaHandle, int jobid, BSA_UInt32 length, char **jobInfo)

DESCRIPTION

NBBSAGetJobInfo() returns information about a job. The information obtained is equivalent to the information that is displayed by the following command:

```
bpdbjobs -jobid id -all_columns
```

See the NetBackup Commands Reference Guide for details. NBBSAGetJobInfo() can be called at any time during any transaction to obtain the job information for the job ID specified. The job information is returned in an array of pointers to character strings (for fields that have no information, the pointer is NULL). The NBBSAGetJobInfo() function allocates storage for each character string. After the application has completed its use of the job information, NBBSAFreeJobInfo() should be called to free the allocated storage.

The following #define constants are defined the include file nbbsa.h and should be used to obtain a specific piece of information for the specified job ID:

```
#define NBBSA_JOB_ID 0
#define NBBSA_JOB_TYPE 1
#define NBBSA_JOB_STATE 2
#define NBBSA_JOB_STATUS 3
#define NBBSA_JOB_POLICY_NAME 4
#define NBBSA_JOB_SCHEDULE_NAME 5
#define NBBSA_JOB_CLIENT_NAME 6
#define NBBSA_JOB_MEDIA_SERVER 7
#define NBBSA_JOB_START_TIME 8
#define NBBSA_JOB_ELAPSED_TIME 9
#define NBBSA_JOB_END_TIME 10
#define NBBSA_JOB_STORAGE_UNIT 11
#define NBBSA_JOB_NUMBER_OF_TRIES 12
#define NBBSA_JOB_OPERATION 13
#define NBBSA_JOB_DATA_WRITTEN 14
#define NBBSA_JOB_FILES_WRITTEN 15
#define NBBSA_JOB_LAST_WRITTEN_PATH 16
#define NBBSA_JOB_PERCENT_COMPLETE 17
#define NBBSA_JOB_PID 18
```

```
#define NBBSA_JOB_USER_ACCOUNT 19
#define NBBSA_JOB_SUBTYPE_CODE 20
#define NBBSA_JOB_POLICY_TYPE 21
#define NBBSA_JOB_SCHEDULE_TYPE 22
#define NBBSA_JOB_CONFIGURED_PRIORITY 23
#define NBBSA_JOB_SERVER_GROUP_NAME 24
#define NBBSA_JOB_MASTER_SERVER_NAME 25
#define NBBSA_JOB_RETENTION_LEVEL 26
#define NBBSA_JOB_RETENTION_PERIOD 27
#define NBBSA_JOB_COMPRESSION 28
#define NBBSA_JOB_ESTIMATED_KILOBYTES 29
#define NBBSA_JOB_ESTIMATED_FILES 30
#define NBBSA_JOB_FILE_LIST_COUNT 31
#define NBBSA_JOB_FILE_PATHS_WRITTEN 32
#define NBBSA_JOB_TRY_COUNT 33
#define NBBSA_JOB_TRY_PID 34
#define NBBSA_JOB_TRY_STORAGE_UNIT 35
#define NBBSA_JOB_TRY_SERVER 36
#define NBBSA_JOB_TRY_START_TIME 37
#define NBBSA_JOB_TRY_ELAPSED_TIME 38
#define NBBSA_JOB_TRY_END_TIME 39
#define NBBSA_JOB_TRY_STATUS_CODE 40
#define NBBSA_JOB_TRY_STATUS_DESCRIPTION 41
#define NBBSA_JOB_TRY_STATUS_LINES_COUNT 42
#define NBBSA_JOB_TRY_STATUS_LINES 43
#define NBBSA_JOB_TRY_DATA_WRITTEN 44
#define NBBSA_JOB_TRY_FILES_WRITTEN 45
#define NBBSA_JOB_PARENT_JOB_NUMBER 46
#define NBBSA_JOB_TRANSFER_SPEED 47
#define NBBSA_JOB_COPY_NUMBER 48
#define NBBSA_JOB_ROBOT 49
#define NBBSA_JOB_VAULT_ID 50
#define NBBSA_JOB_VAULT_PROFILE 51
#define NBBSA_JOB_VAULT_SESSION 52
#define NBBSA_JOB_TAPES_TO_EJECT 53
#define NBBSA_JOB_SOURCE_STORAGE_UNIT 54
#define NBBSA_JOB_SOURCE_MEDIA_SERVER 55
#define NBBSA_JOB_SOURCE_MEDIA_ID 56
#define NBBSA_JOB_DESTINATION_MEDIA_ID 57
#define NBBSA_JOB_STREAM_NUMBER 58
#define NBBSA_JOB_SUSPENDABLE 59
#define NBBSA_JOB_RESUMABLE 60
#define NBBSA_JOB_RESTARTABLE 61
```

```
#define NBBSA_JOB_DATA_MOVEMENT_TYPE 62
#define NBBSA_JOB_SNAPSHOT_OPERATION 63
#define NBBSA_JOB_BACKUP_ID 64
#define NBBSA_JOB_KILLABLE 65
#define NBBSA_JOB_CONTROLLING_HOST 66
#define NBBSA_JOB_OFF_HOST_TYPE 67
#define NBBSA_JOB_FIBER_TRANSPORT_USAGE 68
#define NBBSA_JOB_QUEUE_REASON 69
#define NBBSA_JOB_OPTIONAL_REASON 70
#define NBBSA_JOB_DEDUP_RATIO 71
#define NBBSA_JOB_ACCELERATOR_OPTIMIZATION 72
#define NBBSA_JOB_INSTANCE_DATABASE_NAME 73
#define NBBSA_JOB_LAST_FIELD 73

  The file paths for the NBBSA_JOB_FILE_PATHS_WRITTEN string will
  be delimited by a newline character.
  The status lines for the NBBSA_JOB_TRY_STATUS_LINES string will
  be delimited by a newline character.

--- Sample code ---

char *job_info[NBBSA_JOB_LAST_FIELD+1];

Rc=NBBSAGetJobInfo(BsaHandle,job_id,NBBSA_JOB_LAST_FIELD+1,job_info);
if (Rc == BSA_RC_SUCCESS) {
    printf("Job Status: %s\n", job_info[NBBSA_JOB_STATUS]);

    Rc=NBBSAFreeJobInfo(job_info);
}
```

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
int jobId (I)	The job ID for which you want to obtain information.
BSA_UInt_32 length(I)	The length of the jobInfo parameter that is an array of pointers to the character strings. The length of this array should be NBBSA_JOB_LAST_FIELD+1.

char **JobInfo(O)	The job information is returned in this array of pointers to character strings. A specific piece of information can be obtained by indexing the arrays using one of the #define constants listed in the description. If there is no information for a given field, the pointer is NULL.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_BUFFER_TOO_SMALL	The array pointed to by the JobInfo parameter is not large enough. The length parameter specifies the length of the array and it should be `NBBSA_JOB_LAST_FIELD+1`.
BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation was aborted.
BSA_RC_INVALID_HANDLE	The handle that was used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_SUCCESS	The function succeeded.

NBBSAGetMediaIds

Obtain media IDs for a NetBackup image.

SYNOPSIS

#include <nbbsa.h>

int NBBSAGetMediaIds(BSA_Handle bsaHandle, BSA_UInt64 copyId, BSA_UInt32 *sizePtr, char *MediaIdsPtr)

DESCRIPTION

NBBSAGetMediaIds() returns a null delimited text string containing the media IDs that are associated with a NetBackup image given the copyId of any XBSA object in the image. The value for copyId can be obtained from a previous BSAQueryObject() call. The list of null delimited media IDs is terminated with an empty string. This can also be called a double null terminated string.

Examples are as follows:

- Image with two media IDs: `MediaId1\0MediaId2\0\0`
- Image with one media ID: `MediaId3\0\0`

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
BSA_UInt64 copyId (I)	The unique ID of an XBSA Object that is contained in the image being queried for media IDs. The value for a specific XBSA object can be obtained through a BSAQueryObject() call.
BSA_UInt32 *sizePtr (I/O)	Pointer to the size, in characters, of the buffer pointed to by the MediaIdsPtr parameter. Returns the size of the string pointed to by MediaIdsPtr.
char * MediaIdsPtr (O)	Pointer to a buffer that receives a null delimited character string of media IDs.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_BUFFER_TOO_SMALL	The buffer pointed to by MediaIdsPtr is not large enough. The buffer size, in characters, required to hold the media ID string and its terminating null character is stored in the location pointed to by sizePtr.
BSA_RC_NULL_ARGUMENT	There was a NULL pointer in one of the arguments.
BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Cannot access the requested image. Cannot retrieve the media IDs with the given copyId.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect. Within a transaction calling NBBSAGetMediaIds(), you can embed BSAQueryObject() and BSAGetNextQueryObject() calls. This allows the XBSA application to intermix get NBBSAGetMediaIds() calls with BSAQueryObject() and BSAGetNextQueryObject() calls to obtain the media IDs of multiple images within one transaction. Backup and restore operations are not allowed within a transaction that calls NBBSAGetMediaIds().
BSA_RC_INVALID_COPYID	The copyId field cannot be zero.
BSA_RC_INVALID_HANDLE	The handle that is used to associate this call with a previous BSAInit() call is invalid.
BSA_RC_OBJECT_NOT_FOUND	The given copyId does not exist.

BSA_RC_SUCCESS	The function succeeded.

NBBSAGetMultipleObjects

Initiate a restore of a list of objects

SYNOPSIS

#include <nbbsa.h>

int NBBSAGetMultipleObjects(BSA_Handle bsaHandle, NBBSA_DESCRIPT_LIST * descriptList)

DESCRIPTION

NBBSAGetMultipleObjects() prepares the NetBackup XBSA interface for retrieving the data of multiple XBSA objects that are from the same backup image. It validates the descriptor list, checking that all of the copyId's are valid, that all of the objects are part of the same image, and that the object descriptors are in the correct order. Then, it initiates a connection with the NetBackup server to start the retrieval process for the objects. If any of the objects don't exist, the operation is aborted. When the multiple object restore has been started, the objects can be retrieved in order using the BSAGetObject() - BSAGetData() - BSAEndData() calls.

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
NBBSA_DESCRIPT_LIST *descriptList	Pointer to a list of objectDescriptors that are to be retrieved.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_ABORT_SYSTEM_ERROR	System detected error, operation aborted.
BSA_RC_ACCESS_FAILURE	Access to the requested object is not possible. Cannot retrieve object with given copyId.
BSA_RC_INVALID_CALL_SEQUENCE	The sequence of API calls is incorrect.
BSA_RC_INVALID_COPYID	A value in the copyId is invalid.
BSA_RC_INVALID_HANDLE	The handle that is used to associate this call with a previous BSAInit() call is invalid.

BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_OBJECT_NOT_FOUND	The given copyId does not exist.
BSA_RC_SUCCESS	The multiple object restore has successfully been initiated.

NBBSAGetServerError

Get the error code and text from the NetBackup server.

SYNOPSIS

#include <XBSA.h>

#include <nbbsa.h>

int NBBSAGetServerError(BSA_Handle bsaHandle, int *ServerStatus, BSA_UInt32 sizePtr, char *ServerStatusStr)

DESCRIPTION

NBBSAGetServerError returns the error code and corresponding text message generated from the NetBackup processes. This can be useful in logging a more accurate cause of a failure as compared to the NBBSA error code, which tends to be very generic when the error occurred on the NetBackup server.

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous call to BSAInit.
int *ServerStatus (O)	Pointer to the NetBackup error code that has been returned from the NetBackup server.
BSA_UInt32 sizePtr (I/O)	Pointer to the size of the ServerStatusStr in bytes.
char *ServerStatusStr (O)	Pointer to the text string of the server status.

EXTENDED DESCRIPTION

NBBSAGetServerError requires the ServerStatusStr string to be allocated and the size of this string to be entered in the sizePtr parameter. This ensures that the NetBackup error text can fit in the string. The function resets the sizePtr to the actual size of the error text that is returned.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_BUFFER_TOO_SMALL	The size of the data buffer is too small for the error text.
BSA_RC_NULL_ARGUMENT	A NULL pointer was encountered in one of the arguments.
BSA_RC_SUCCESS	The function successfully returned the error.

NBBSALogMsg

Log a message to the XBSA logs.

SYNOPSIS

#include <nbbsa.h>

int NBBSALogMsg(BSA_Handle bsaHandle, int msgType, char *msgBuf, char *callingFunc)

DESCRIPTION

NBBSALogMsg() gives the XBSA application the ability to log messages to the same debug log file that is being used by the NetBackup XBSA interface with the log messages being the same format. If used correctly, this can make debugging easier because you can follow the complete flow of the combined XBSA application and XBSA interface.

The log file is opened in BSAInit(), so logging cannot occur until the session has been initiated. The log file is closed in BSATerminate().

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
int msgType (I)	The level of error that is displayed with the timestamp and message.
char *msgBuf (I)	The text of the error message.
char *callingFunc (I)	The function name that is calling this function. It is displayed in the log file.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_SUCCESS	The function succeeded.

NBBSASetEnv

Set the value of a single XBSA environment value.

SYNOPSIS

#include <nbbsa.h>

int NBBSASetEnv(BSA_Handle bsaHandle, char *EnvVar, char *EnvVal)

DESCRIPTION

NBBSASetEnv() gives the XBSA application the ability to set the value of a specific XBSA environment variable after the beginning of a session. If the variable does not exist in the environment, it is added. If the variable exists in the environment, the value is replaced. Some of the XBSA environment variables can only be set or updated at certain points in the session. See "Environment variable definitions" on page 25. If the variable cannot be set or updated, the original value remains.

The XBSA specifications do not provide a way for these XBSA environment variables to be reset after the session has been initiated with BSAInit().

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
char *EnvVar (I)	Pointer to a null-terminated string that specifies the XBSA environment variable whose value is being set.
char *EnvVal (I)	Pointer to a null-terminated string containing the new value of the specified XBSA environment variable. If this parameter is NULL, the variable is deleted from the current sessions XBSA environment.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_SUCCESS	The specified XBSA environment variable has been set.
BSA_RC_NULL_ARGUMENT	A required argument was passed as a NULL pointer.
BSA_RC_ABORT_SYSTEM_ERROR	Unable to increase the size of the session's XBSA environment block.
BSA_RC_INVALID_ENV	The variable is not a supported environment variable.

NBBSAUpdateEnv

Update the current environment for the session.

SYNOPSIS

#include <nbbsa.h>

int NBBSAUpdateEnv(BSA_Handle *bsaHandle, char **envPtr)

DESCRIPTION

NBBSAUpdateEnv() resets the environment pairs in the current environment. It performs the same functionality as NBBSASetEnv() except it takes a string of multiple (keyword, value) pairs. The same restrictions apply to updating some of the restricted variables. If a variable exists in the environment but is not included in the list being updated, it remains in the environment.

The XBSA specifications do not provide a way for these XBSA environment variables to be reset after the session has been initiated with BSAInit().

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
char **envPtr (I)	Pointer to a structure that contains the new environment variables (keyword, value) pairs, for the session. The environment consists of a pointer to an array of strings.

RETURN VALUE

The following return codes are returned by this function:

BSA_RC_SUCCESS	The specified XBSA environment variable has been set.

NBBSAValidateFeatureId

Validate the license key for the specified feature ID.

SYNOPSIS

#include <nbbsa.h>

int NBBSAValidateFeatureId(BSA_Handle bsaHandle, char * featureIdList, int validationOption)

DESCRIPTION

NBBSAValidateFeatureId() parses the featureIdList string for the list of feature IDs that need to be validated. If MATCH_ANY_FEATURE_ID is specified as the

validationOption, BSA_RC_SUCCESS is returned if a license key exists for any of the feature IDs in the list. If MATCH_ALL_FEATURE_ID is specified as the validationOption, BSA_RC_SUCCESS is returned if a license key exists for all of the feature IDs in the list.

PARAMETERS

BSA_Handle bsaHandle (I)	The handle that associates this call with a previous BSAInit() call.
char *featureIdList (I)	This parameter is a space delimited list of the license feature id(s) that are to be validated.
int validationOption (I)	Specifies which combination of features needs to exist to be valid. Currently supports MATCH_ANY_FEATURE_ID or MATCH_ALL_FEATURE_ID.

RETURN VALUE

The following return codes are returned by this function:

NBBSA_RC_FEATURE_NOT_LICENSED	The feature does not have a valid license.
BSA_RC_SUCCESS	The specified feature id(s) have a valid license.

Type definitions

The following type definitions are provided for use within the NetBackup XBSA interfaces.

Table 6-4 XBSA Type Definitions

Data Type	Type Name	Example Type Definition
16-bit Integer	BSA_Int16	typedef short BSA_Int16;
32-bit Integer	BSA_Int32	typedef int BSA_Int32;
64-bit Integer	BSA_Int64	typedef struct { BSA_Int32 left; BSA_Int32 right; } BSA_Int64;

Table 6-4 XBSA Type Definitions *(continued)*

Data Type	Type Name	Example Type Definition
16-bit Unsigned Integer	BSA_UInt16	typedef unsigned short BSA_UInt16;
32-bit Unsigned Integer	BSA_UInt32	typedef unsigned int BSA_UInt32;
64-bit Unsigned Integer	BSA_UInt 64	typedef struct {
		BSA_UInt32 left;
		BSA_UInt32 right;
		} BSA_UInt64;
Shared Memory Buffer reference	BSA_ShareId	<Not_Used>

Enumerated types

The following enumerated type definitions are provided for use within the NetBackup XBSA interfaces. For enumerations used in queries, the value 1 is reserved for use as a wildcard (ANY) value.

BSA_CopyType

The BSA_CopyType enumeration describes the type of the operation used to create a NetBackup XBSA object. It is defined as follows:

```
typedef enum {
       BSA_CopyType_ANY = 1,
       BSA_CopyType_ARCHIVE = 2,
       BSA_CopyType_BACKUP = 3
} BSA_CopyType;
```

The meaning of the enumeration values is as follows:

Table 6-5 BSA_CopyType Enumeration Values

Constant	Definition	Value
ANY	1	Used for matching any copy type (for example, "backup" or "archive" in the copy type field of structures for selecting query results).

Table 6-5 BSA_CopyType Enumeration Values *(continued)*

Constant	Definition	Value
ARCHIVE	2	Specifies that the copy type should be "archive."
BACKUP	3	Specifies that the copy type should be "backup."

BSA_ObjectStatus

The BSA_ObjectStatus enumeration describes the current status of the NetBackup XBSA object. It is defined as follows:

```
typedef enum {
       BSA_ObjectStatus_ANY = 1,
       BSA_ObjectStatus_MOST_RECENT = 2,
       BSA_ObjectStatus_NOT_MOST_RECENT = 3
} BSA_ObjectStatus;
```

The meaning of the enumeration values is as follows:

Table 6-6 BSA_ObjectStatus Enumeration Values

Constant	Value	Definition
ANY	1	Provides a wildcard function. Can only be used in queries.
MOST_RECENT	2	Indicates that this is the most recent backup copy of an object.
NOT_MOST_RECENT	3	Indicates that this is not the most recent backup copy, or that the object itself no longer exists.

BSA_ObjectType

The BSA_ObjectType enumeration describes the original data type of the object. It is defined as follows:

```
typedef enum {
       BSA_ObjectType_ANY  = 1,
       BSA_ObjectType_FILE    = 2,
       BSA_ObjectType_DIRECTORY = 3,
       BSA_ObjectType_OTHER = 4
} BSA_ObjectType;
```

The meaning of the enumeration values is as follows:

Table 6-7 BSA_ObjectType Enumeration Values

Constant	Value	Definition
ANY	1	Used for matching any object type (for example, "file" or directory") value in the object type field of structures for selecting query results.
FILE	2	Used by the application to indicate that the type of application object is a "file" or single object.
DIRECTORY	3	Used by the application to indicate that the type of application object is a "directory" or container of objects.
OTHER	4	Used by the application to indicate that the type of application object is neither a "file" nor a "directory".

BSA_Vote

The BSA_Vote enumeration describes whether or not the transaction is to be committed. It is defined as follows:

```
typedef enum {
       BSA_Vote_COMMIT = 1,
       BSA_Vote_ABORT  = 2
       NBBSA_Vote_CONDITIONAL = 99
} BSA_Vote;
```

The meaning of the enumeration values is as follows:

Table 6-8 BSA_Vote Enumeration Values

Constant	Value	Definition
COMMIT	1	The transaction is to be committed.
ABORT	2	The transaction is to be aborted.
CONDITIONAL	99	The transaction is to be committed, report only conditional success.

Constant values

The following constants are defined for use in the NetBackup XBSA interfaces:

Table 6-9 XBSA Constant Values

Constant	Value	Definition
BSA_ANY	1	General-purpose enumeration wildcard value
BSA_MAX_APPOBJECT_OWNER	64	Max end-user object owner length
BSA_MAX_BSAOBJECT_OWNER	64	Max BSA object owner length
BSA_MAX_DESCRIPTION	100	Description field
BSA_MAX_OBJECTSPACENAME	1024	Max ObjectSpace name length
BSA_MAX_OBJECTINFO	256	Max object info size
BSA_MAX_PATHNAME	1024	Max path name length
BSA_MAX_RESOURCETYPE	31	Max resourceType name length
BSA_MAX_TOKEN_SIZE	64	Max size of a security token

Data structures

The following data structures are provided for use within the NetBackup XBSA interfaces.

BSA_ApiVersion

The BSA_ApiVersion structure describes the version of the API that is implemented. It is defined as follows:

```
typedef struct {
       BSA_UInt16                  issue;
       BSA_UInt16                  version;
       BSA_UInt16                  level;
} BSA_ApiVersion;
```

The usage of the structure fields is defined as follows:

Table 6-10 BSA_ApiVersion Structure Fields

Field	Description
issue	Issue number of the XBSA specification
version	Version number of the XBSA specification

Table 6-10 BSA_ApiVersion Structure Fields *(continued)*

Field	Description
level	NetBackup XBSA-defined version number

The NetBackup XBSA interface is an implementation of the XBSA Technical Standard (document - C425); the values should be 1,1,0.

BSA_DataBlock32

The BSA_DataBlock32 structure is used to pass data between an XBSA application and the NetBackup XBSA interface. It is defined as follows:

```
typedef struct {
       BSA_UInt32  bufferLen;
       BSA_UInt32  numBytes;
       BSA_UInt32  headerBytes;
       BSA_ShareId shareId;
       BSA_UInt32  shareOffset;
       void       *bufferPtr
} BSA_DataBlock32;
```

The usage of the structure fields is defined as follows:

Table 6-11 BSA_DataBlock32 Structure Fields

Field Name	Definition
bufferLen	Length of the allocated buffer
numBytes	Actual number of bytes read from or written to the buffer, or the minimum number of bytes needed
headerBytes	Number of bytes used at start of buffer for header information (offset to data portion of buffer)
shareId	Value used to identify a shared memory block.
shareOffset	Specifies the offset of the buffer in the shared memory block.
bufferPtr	Pointer to the buffer

The values assigned to the various structure fields would always obey the following relationships:

bufferLen >= headerBytes + numBytes

trailerBytes == (bufferLen - numBytes - headerBytes)

The header and trailer portions of the buffer are reserved for the use of the NetBackup XBSA interface, and should not be modified by the XBSA application. The XBSA application should only write to the data portion of the buffer, which is the only portion used for transferring application data.

The sizes for the header and trailer portions of the buffer that are required by the NetBackup XBSA interface are obtained by calling BSACreateObject() or BSAGetObject().

BSA_ObjectDescriptor

The BSA_ObjectDescriptor structure is used to describe an object. It is defined as follows:

#include <time.h>

```
typedef struct {
    BSA_UInt32           rsv1;
    BSA_ObjectOwner      objectOwner;
    BSA_ObjectName       objectName;
    struct tm            createTime;
    BSA_CopyType         copyType;
    BSA_UInt64           copyId;
    BSA_UInt64           restoreOrder;
    char                 rsv2[31];
    char                 rsv3[31];
    BSA_UInt64           estimatedSize;
    char                 resourceType[BSA_MAX_RESOURCETYPE];
    BSA_ObjectType       objectType;
    BSA_ObjectStatus     objectStatus;
    char                 rsv4[31];
    char                 objectDescription[BSA_MAX_DESCRIPTION];
    unsigned char        objectInfo[BSA_MAX_OBJECTINFO];
} BSA_ObjectDescriptor;
```

Some of the fields in this structure are supplied by the XBSA application (Direction = in), and some by the NetBackup XBSA interface (Direction = out). Some fields are optional.

The usage of the structure fields is defined as follows:

Table 6-12 BSA_ObjectDescriptor Structure Fields

Field Name	Definition	Supplied By	Status
rsv1	Reserved field	-	-

Table 6-12 BSA_ObjectDescriptor Structure Fields *(continued)*

Field Name	Definition	Supplied By	Status
objectOwner	Owner of the object	Application	optional
objectName	Object name	Application	mandatory
createTime	Creates time	Interface	mandatory
copyType	Copies type: archive or backup	Application	mandatory
copyId	Unique object identifier	Interface	mandatory
restoreOrder	Provides hints to the XBSA application that allow it to optimize the order of object retrieval requests	Interface	optional
rsv2	reserved field	-	-
rsv3	reserved field	-	-
estimatedSize	Estimated object size in bytes, can be up to (2>64 - 1) bits	Application	mandatory
resourceType	for example, a UNIX file system	Application	mandatory
objectType	for example, file, directory, or database	Application	mandatory
objectStatus	Most recent / Not most recent	Interface	mandatory
rsv4	reserved field	-	-
objectDescription	Descriptive label for the object	Application	optional
objectInfo	Application-specific information	Application	optional

All values in a BSA_ObjectDescriptor must be valid before the BSA_ObjectDescriptor as a whole is valid. For enumerations valid values exclude the enumeration "ANY". For strings valid values are null-terminated.

The optional string value is the empty string. The optional restoreOrder value is zero. The optional objectInfo value is an empty string.

The mandatory objectName must have a non-empty string in the pathName field. The mandatory createTime must be a valid time in UTC. The mandatory copyId must be non-zero. The mandatory resourceType must have a non-empty string value.

All string values cannot contain any new line, carriage return, or line feed characters. Although this cannot cause an error when the object is being created, the object cannot be restored.

BSA_ObjectName

The BSA_ObjectName structure is the name assigned by an XBSA application to a NetBackup XBSA object. It is defined as follows:

```
typedef struct {
    char  objectSpaceName[BSA_MAX_OBJECTSPACENAME];
    char  pathName[BSA_MAX_PATHNAME];
} BSA_ObjectName;
```

The usage of the structure fields is defined as follows:

Table 6-13 BSA_ObjectName Structure Fields

Field Name	Definition
objectSpaceName	Highest-level name qualifier
pathName	Object name within objectSpaceName

An objectSpaceName is an optionally defined, fixed-length character string. It identifies a logical space, called an object space, to which the object belongs. For example, an object space may be used to identify a storage volume (for example, a disk partition, or a floppy disk), or a database in the XBSA application's domain.

The NetBackup XBSA interface uses the concept of an object space to provide a primary grouping of NetBackup XBSA objects that can be used for an object search by a user and/or for object management. Additional groupings are provided by object attributes. Examples of an objectSpaceName are C: Drive and VolumeLabel=XYZ.

A pathName is a hierarchical character string that identifies a NetBackup XBSA object within an ObjectSpace. While the pathname does not need to correspond to an actual file path, NetBackup requires that the first character is a '/'. This is true for both UNIX and Windows.

An example of a pathName for the backup copy of a UNIX file can be its original path name and file name, for example, `/x/y/z/xyx.c`.

The value of the delimiter that is used to separate the name components can be obtained by calling BSAGetEnvironment().

BSA_ObjectOwner

The BSA_ObjectOwner structure is the name of the owner of an object. It is defined as follows:

```
typedef struct {
    char  bsa_ObjectOwner[BSA_MAX_BSAOBJECT_OWNER];
    char  app_ObjectOwner[BSA_MAX_APPOBJECT_OWNER];
} BSA_ObjectOwner;
```

The usage of the structure fields is defined as follows:

Table 6-14 BSA_ObjectOwner Structure Fields

Field Name	Definition
bsa_ObjectOwner	The name that the NetBackup XBSA interface authenticates
app_ObjectOwner	The name defined by the application

In NetBackup XBSA Version 1.1.0, the actual object owner is determined between the NetBackup XBSA interface and NetBackup. If the XBSA application specifies the bsa_ObjectOwner, the value is stored with the object or validated against it, but it does not define the object ownership. If the object was created by a different user, unless you are a root administrator, you cannot restore the object even if you specify the correct bsa_ObjectOwner.

An app_ObjectOwner is an optional name, such as an actual end-user name, provided by the respective XBSA application, so that the NetBackup XBSA interface can provide assistance to support application-specific access control by optimizing access for the given app_ObjectOwner.

The app_ObjectOwner can have multiple components defined in the application, such as a group name and a user ID. In general, it is a hierarchical character string. An app_ObjectOwner is not registered with the NetBackup XBSA interface. Its registration and authentication is the responsibility of the XBSA application.

Examples of a typical app_ObjectOwner are Smith, AccountingDept.Clerk1 and * (unspecified).

BSA_QueryDescriptor

The BSA_QueryDescriptor structure is used to query the repository to locate objects. It is defined as follows:

#include <time.h>;

```
typedef struct {
   BSA_ObjectOwner       objectOwner;
   BSA_ObjectName        objectName;
   struct tm             createTime_from;
   struct tm             createTime_to;
   struct tm             rsv1;
   struct tm             rsv2;
   BSA_CopyType          copyType;
   char                  rsv3[31];
   char                  rsv4[31];
   char                  rsv5[31];
   BSA_ObjectType         objectType;
   BSA_ObjectStatus      objectStatus;
   char                  rsv6[100];
} BSA_QueryDescriptor;
```

The usage of the structure fields is defined as follows:

Table 6-15 BSA_QueryDescriptor Structure Fields

Field Name	Definition
objectOwner	Owner of the object
objectName	Object name
createTime_from	Date time to start looking for the object
createTime_to	Date time to stop looking for the object
rsv1	reserved field
rsv2	reserved field
copyType	Copy type: archive or backup
rsv4	reserved field

Table 6-15 BSA_QueryDescriptor Structure Fields *(continued)*

Field Name	Definition
rsv5	reserved field
rsv5	reserved field
objectType	Examples are file, directory, database
objectStatus	Most recent / not most recent
rsv8	reserved field

BSA_SecurityToken

The BSA_SecurityToken structure contains an application-specific security token. It is defined as follows:

typedef char BSA_SecurityToken[BSA_MAX_TOKEN_SIZE];

Chapter 7

Process flow and troubleshooting

This chapter includes the following topics:

About Process flow and troubleshooting

The XBSA interface is provided to insulate the XBSA applications from knowing about the internals of NetBackup and the processes and calls that are required to do backups and restores. This is appropriate when the application is working correctly. In the event that a problem occurs, this chapter gives a brief description of the NetBackup processes that get instantiated for each backup and restore. If process fails, a log is produced that contains information pertaining to the failure. All logs should be examined, however, for the root cause of the failure.

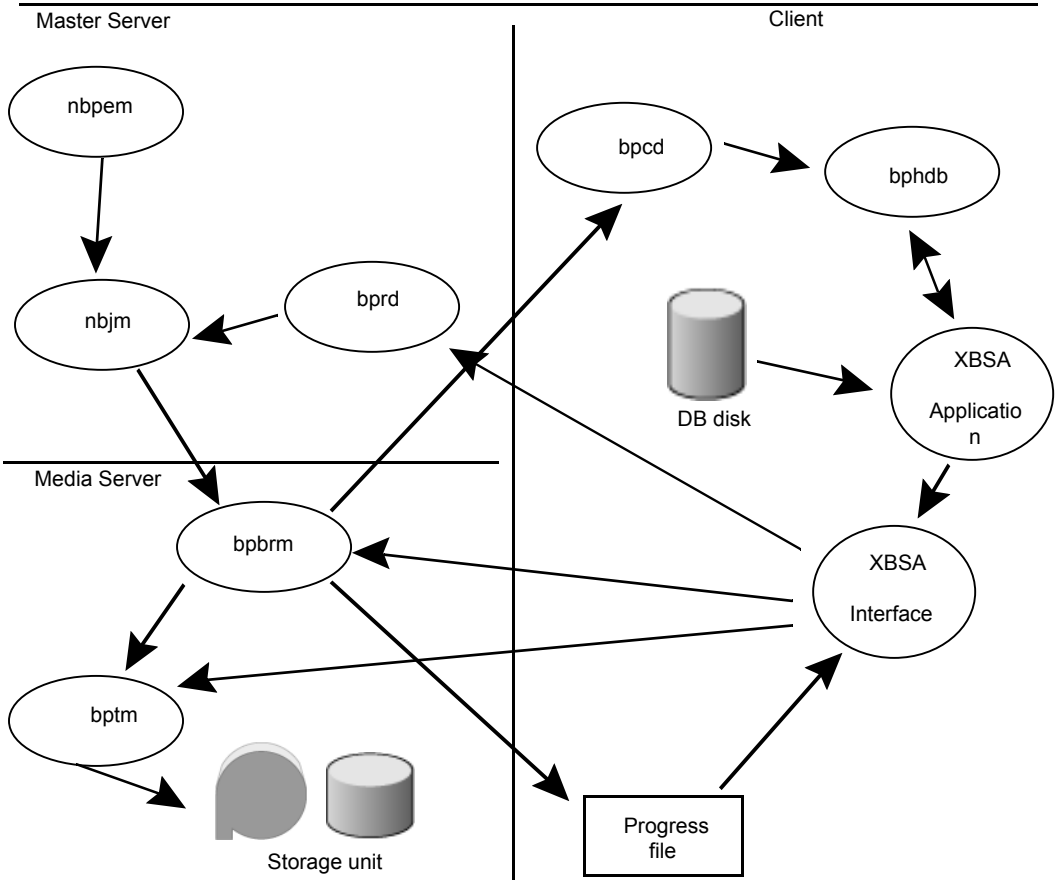
Backup

The backup stream diagram contains the processes involved in a backup being executed through a scheduled backup. bphdb is called to initiate the XBSA application, which then initiates one or more user-directed jobs.

If the backup is initiated from the XBSA application, it starts at that point. In the diagram below, the processes are divided among three logical machines, the Master Server, Media Server, and Client, but they can exist on only one or two machines.

See the backup process diagrams in the NetBackup Logging Reference Guide.

Figure 7-1 Stream backup process flow diagram



Stream backup process flow description

This description provides a basic process flow. There are other processes involved that are not explicitly specified here. These processes include bpcd and some parent/child processes. Most connectivity from the server to the client initially goes through bpcd, such as initiating bphdb and writing to the progress log. And many of the processes, such as bpbrm, bptm, and so on, initiate child processes. To keep the description simple, these processes are not included in the process steps.

Stream backup procedure

The stream backup procedure is as follows:

1. nbpem determines that a backup is scheduled to run and it initiates a backup job by nbjm.
2. nbjm starts bpbrm, which makes a request by bpcd.
3. bpbrm makes a request by the bpcd daemon to start bphdb on the client.
4. bphdb executes the backup script (which is contained in the Backup Selections list of the backup policy). Bphdb waits for an exit status from this script so that it can pass a status back to the server.
5. The backup script initiates the backup utility of the XBSA application.

   (If it is not a scheduled backup operation, but is initiated on the client by the XBSA application, then the backup process starts here.)
6. The application initiates the XBSA interface by starting one or more sessions. Each session should be started in its own process. In this diagram, we assume that there is only one stream. In reality, each stream follows each of these steps.
7. The backup is initiated with the first call to BSACreateObject(). This causes the XBSA interface to make a bprd request to initiate a backup.
8. bprd submits a backup request to nbpem, which submits a job for nbjm. If this was a scheduled backup, there are now two backup jobs. nbjm initiates a bpbrm and a bpdbm process.
9. bpbrm initiates a bptm/bpdm process (bptm if tape storage unit, bpdm if disk storage unit). bptm initiates the process to mount media.
10. bpbrm writes progress information to the progress file on the client (by bpcd). This information includes sockets, status, backup attributes, and so on.
11. XBSA reads the progress file to find the sockets and other information and connects to bpbrm on the name socket. It continues to read the progress file until it gets the message that it can continue the backup.
12. XBSA connects to bptm/bpdm through shared memory (if applicable) or on the data socket if the client and media server are separate machines.
13. XBSA sends the XBSA object entry to bpbrm, which sends it on to bpdbm to be catalogued.
14. At this point, BSACreateObject() returns to the XBSA application. XBSA is ready to receive data.
15. The application fills buffers and calls BSASendData() to have the XBSA interface send these buffers to bptm/bpdm through the established connection.
16. bptm/bpdm writes this data to media or disk storage.

17 When the application has sent all of the data, it indicates this with a BSAEndData() call. XBSA recognizes that the object is complete.

18 The XBSA application can then call BSACreateObject() again to create more objects. The subsequent CreateObject calls do not cause new jobs or connections, but continue with the existing processes.

19 When the XBSA application has completed creating objects, a call to BSAEndTxn() causes XBSA to initiate the termination process. XBSA sends a client status to bpbrm, that terminates the server processes (bptm).

20 bpbrm writes the server status to the progress log and is read by XBSA. This allows the XBSA interface to confirm that the image has been successfully catalogued and all of the data was written to media.

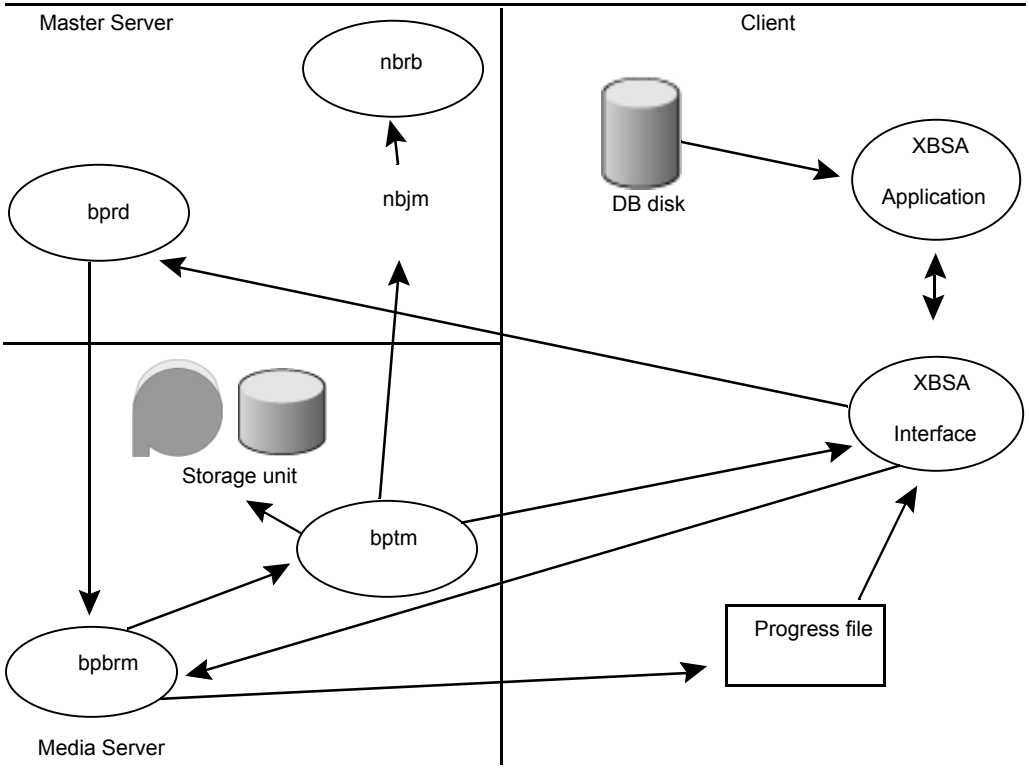
21 XBSA then passes this status back to the application through the return status from BSAEndTxn(). This status is passed back to bphdb, that passes it back to the original scheduled job to complete the backup. This status displays in the Activity Monitor for the originally scheduled job.

Restore

The Restore diagram contains the processes involved in a restore that is executed from the XBSA application. The restore process is very similar to the backup process except the flow of data is reversed. While it is possible to run a scheduled restore through bphdb, this is not a standard procedure; this diagram starts with the XBSA application initiating the restore.

See the restore process diagrams in the NetBackup Logging Reference Guide.

Figure 7-2 Stream restore process flow diagram



Stream restore process flow description

Similar to the backup diagram, the restore diagram is simple, without some of the parent/child processes. The query process, that can be included in a restore operation, is not described in the following process. This process assumes that the XBSA application has the object descriptor to be passed to BSAGetObject.

Stream restore procedure

The stream restore procedure is as follows:

1. The XBSA application initiates the XBSA interface by starting one or more sessions. Each session is started in its own process. For simplicity, we assume that there is only one stream in this diagram. In reality, each stream has each of these steps.
2. The restore is initiated with the first call to BSAGetObject(). This causes the XBSA interface to make a bprd request to initiate a restore.

3. bprd initiates a bpbrm process.
4. bpbrm initiates a bptm/bpdm process (bptm if tape storage unit, bpdm if disk storage unit). bptm gets the resources from nbjm/nbrb and initiates the process to mount media and start reading the data.
5. bpbrm writes progress information to the progress file on the client (by bpcd). This information includes sockets, status, restore attributes, and so on.
6. XBSA reads the progress file to find the sockets and other information and connects to bpbrm on the name socket. It continues to read the progress file until it gets the message that it can continue the restore.
7. XBSA connects to bptm/bpdm by shared memory (if applicable) or on the data socket if the client and media server are separate machines.
8. At this point, BSAGetObject() returns to the XBSA application. XBSA is ready to receive data.
9. The application passes buffers to BSASendData() to have the XBSA interface fill these buffers with data from bptm/bpdm by the established connection.
10. bptm/bpdm continues to read this data from media or disk storage and write it to the buffers.
11. When the application has received all data, it indicates this with a BSAEndData() call. XBSA verifies that all of the data from the object has been sent. XBSA sends a client status to bpbrm that terminates the server processes (bptm).
12. bpbrm writes the server status to the progress log and is read by XBSA. This allows the XBSA interface to confirm that the server has successfully read all the data and terminated.
13. The restore has been completed at this time. A call to BSAEndTxn() is required to close the transactions, but other than some internal cleanup, it does not provide any function for restores.

Chapter 8

How to use the sample files

This chapter includes the following topics:

What the sample files do

Included in the SDK are some simple sample programs and scripts. The sample programs can be used as examples of how to use the XBSA functions to create an XBSA application. The sample scripts are examples of how an XBSA application can be executed from a NetBackup schedule.

Sample programs

The SDK includes some simple sample programs that can be used as an example of the sequence of function calls that are required to create new objects, query the NetBackup database for existing objects, retrieve the objects, and delete objects. There is a separate program for each of these functions, although this is for the convenience of the samples and not necessarily a recommended way of building an XBSA application.

These programs cannot run as installed. First, they need to be modified to set the correct hostname of the NetBackup server. Then, they can be compiled and each can be individually run. Below is the description of the programs and what to expect from them if they have not been modified other than setting the hostname.

The following section of the sample programs needs to be modified. The entries ‘server_host’, ‘sample_policy’, and ‘sample_schedule’ need to be replaced with actual values from your environment. These three entries can also be eliminated so that the sample program uses the default values from the NetBackup configuration.

```
/* Populate the XBSA environment variables for this session. */

strcpy(envx[0], "BSA_API_VERSION=1.1.0");
strcpy(envx[1], "BSA_SERVICE_HOST=server_host");
strcpy(envx[2], "NBBSA_POLICY=sample_policy");
strcpy(envx[3], "NBBSA_SCHEDULE=sample_schedule");
envx[4] = NULL;
```

Backup

This program creates one small object. The unique identifier, copyId, is printed out along with the number of bytes that were backed up.

```
copyId: 1 - 1018898698
Bytes backed up: 154
```

Restore

This program retrieves the last object that was created. The copyId is printed out along with the text of the object data and the number of bytes that were retrieved.

```
Retrieving copyId: 1 - 1018898698
This is the sample data that is contained in the sample object that
is being backed up for the purposes of showing how data can be
backed up and restored.
Total bytes retrieved: 154
```

Query

This program searches for all of the objects created by the Backup program. The copyId of each of these objects is printed out.

```
copyId: 1 - 1018898698
copyId: 1 - 1018898638
```

Delete

This program deletes the last object that was created. The copyId of the object being deleted is printed out.

```
Deleting copyId: 1 - 1018898698
```

Sample scripts

Also included are some examples of scripts that can be used to initiate an XBSA application as a scheduled NetBackup job. Again these are very simple scripts based on the sample programs. There are sample scripts for UNIX platforms (`*.sh`) and for Windows platforms (`*.cmd`).

In general use, the XBSA application would have parameters or use system environment variables to communicate the parameters about the backup or restore operations.

See "Running a NetBackup XBSA application" on page 74.

Description of sample files

This section includes a description of the sample files provided with the SDK. All sample files are located in `~sdk/DataStore/XBSA/samples`.

Table 8-1 Description of Sample Files

Filename	Description
Backup.c	This is an example of the functions needed to create an XBSA object.
Query.c	This is an example of the functions needed to search for an XBSA object.
Restore.c	This is an example of the functions needed to retrieve an XBSA object.
Delete.c	This is an example of the functions needed to delete an XBSA object.
Makefile.unix	This is an example Makefile that can be used to compile the sample programs on the UNIX platforms.
Makefile.nt	This is an example Makefile that can be used to compile the sample programs on Windows platforms.

Table 8-1 Description of Sample Files *(continued)*

Filename	Description
backup_script.cmd	This is an example of the script that runs an XBSA application from a NetBackup schedule on a Windows platform.
restore_script.cmd	This is an example of the script that runs an XBSA application from a NetBackup schedule on a Windows platform.
backup_script.sh	This is an example of the script that runs an XBSA application from a NetBackup schedule on a UNIX platform.
restore_script.sh	This is an example of the script that runs an XBSA application from a NetBackup schedule on a UNIX platform.

How to build the sample programs

Also included with the samples are a Makefile for UNIX platforms, Makefile.unix, and one for Windows, Makefile.nt. The Makefiles compile the four sample programs using basic compiler options.

The UNIX Makefile needs to be modified to select which library to use. Library paths for all of the supported platforms are in the Makefile but commented out. The library for the required operating system needs to be chosen along with whether to use an archive library or a shared library.

The following lines are from `Makefile.unix`. One of the CFLAGS and one of the LIBS definitions need to be uncommented. The default is to compile 64 bit using the dynamic shared libraries.

The CFLAGS definitions are compile options. Select a CFLAGS definition for the system that is being compiled on. Note that this is a very minimal set of options and you can add other compile options based on your environment.

```
# Uncomment the CFLAGS for the environment that is being compiled

# Solaris sparc 64 bit
#CFLAGS = -xarch=v9

# Solaris Opteron 64 bit
#CFLAGS = -xtarget=opteron -xarch=generic64

# HP IA64 bit
#CFLAGS = -Ae +DSitanium2 +DD64
```

```
# AIX 64 bit
#CFLAGS = -q64

# Linux x86 64 bit
#CFLAGS = -m64

# Linux on Power PC 64 bit
#CFLAGS = -m64

DEFINES =
INCLUDES= -I$(XBSA_SDK_DIR)/include
```

The LIBS definitions define to which XBSA library to link. A shared object library is installed in `/usr/openv/lib` on all NetBackup clients and can be used for dynamic linking. An archive library for each platform is included in the SDK and can be used to statically link the application. Select a LIBS definition for the system that is being compiled.

```
# Use one of these LIBS to bind dynamically

LIBS = -L/usr/openv/lib -lxbsa -lnbclientcST -lnbbasecST -lnbtlscST -lnbsslST
#LIBS = -L/usr/openv/lib -lxbsa64 -lnbclientcST -lnbbasecST -lnbtlscST -lnbsslST

# Or choose the correct LIBS for your system to bind statically

#LIBS    = $(XBSA_SDK_DIR)/lib/HP-UX-IA64/HP-UX11.31/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/Debian3.10.0/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/RedHat3.10.0/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/SuSE4.4.73/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-ppc64le/IBMpSeriesRedHat3.10.0/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-ppc64le/IBMpSeriesSuSE4.4.21/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-s390x/IBMzSeriesRedHat3.10.0/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux-s390x/IBMzSeriesSuSE4.4.73/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/RS6000/AIX6/libxbsa64.a -ldl -lc
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris10/libxbsa64.a -lintl -lsocket -lnsl
-ldl -ladm
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris_x86_10_64/libxbsa.a -lintl -lsocket
-lnsl -ldl -ladm
```

The Windows Makefile may need to be modified to select which Windows library to use. The Windows Makefile needs to be modified if SDK was installed into a directory other than the default `c:\Program Files`.

```
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows\xbsas.lib
```

Chapter 9

Support and updates

This chapter includes the following topics:

About Support and updates

The NetBackup SDK for DataStore is sold and distributed under specific licensing agreements. These licensing agreements define how the SDK is supported, who to contact for support, and how upgrades are supported. The agreements should also define how an XBSA application is sold and supported with NetBackup. Please review your licensing agreement for details on product support.

Appendix A

Register authorized locations

This appendix includes the following topics:

Registering authorized locations used by a NetBackup database script-based policy

During a backup, NetBackup checks for scripts in the default script location and any authorized locations. The default, authorized script location for UNIX is `usr/openv/netbackup/ext/db_ext` and for Windows is `install_path\netbackup\dbext`. If the script is not in the default script location or an authorized location, the policy job fails. You can move any script into the default script location or any additional authorized location and NetBackup recognizes the scripts. You need to update the policy with the script location if it has changed. An authorized location can be a directory and NetBackup recognizes any script within that directory. An authorized location can also be a full path to a script if an entire directory does need to be authorized.

If the default script location does not work for your environment, use the following procedure to enter one or more authorized locations for your scripts. Use `nbsetconfig` to enter an authorized location where the scripts reside. You can also use `bpsetconfig`, however this command is only available on the master or the media server.

Note: One recommendation is that scripts should not be world-writable. NetBackup does not allow scripts to run from network or remote locations. All scripts must be stored and run locally. Any script that is created and saved in the NetBackup `db_ext` (UNIX) or `dbext` (Windows) location needs to be protected during a NetBackup uninstall.

For more information about registering authorized locations and scripts, review the knowledge base article:

https://www.veritas.com/content/support/en_US/article.100039639

To add an authorized location

1. Open a command prompt on the client.
2. Use `nbsetconfig` to enter values for an authorized location. The client privileged user must run these commands.

   The following examples are for paths you may configure for the Oracle agent. Use the path that is appropriate for your agent.

   - On UNIX:

     ```
     [root@client26 bin]# ./nbsetconfig
     nbsetconfig>DB_SCRIPT_PATH = /Oracle/scripts
     nbsetconfig>DB_SCRIPT_PATH = /db/Oracle/scripts/full_backup.sh
     nbsetconfig>
     <ctrl-D>
     ```

   - On Windows:

     ```
     C:\Program Files\Veritas\NetBackup\bin>nbsetconfig
     nbsetconfig> DB_SCRIPT_PATH=c:\db_scripts
     nbsetconfig> DB_SCRIPT_PATH=e:\oracle\fullbackup\full_rman.sh
     nbsetconfig>
     <ctrl-Z>
     ```

Note: Review the NetBackup Command Reference Guide for options, such as reading from a text file and remotely setting clients from a NetBackup server using `bpsetconfig`. If you have a text file with the script location or authorized locations listed, `nbsetconfig` or `bpsetconfig` can read from that text file. An entry of `DB_SCRIPT_PATH=none` does not allow any script to execute on a client. The `none` entry is useful if an administrator wants to completely lock down a server from executing scripts.

3 (Conditional) Perform these steps on any clustered database or agent node that can perform the backup.

4 (Conditional) Update any policy if the script location was changed to the default or authorized location.

Index