Open Group Technical Standard

Systems Management:

Backup Services API (XBSA)

The Open Group

Open Group Technical Standard

Systems Management: Backup Services API (XBSA)

ISBN: 1-85912-056-3
Document Number: C425

Published in the U.K. by The Open Group, April 1998.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- Consolidating, prioritizing, and communicating customer requirements to vendors
- Conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- Managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- Adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- Licensing and promoting the Open Brand, represented by the ''X'' Device, that designates vendor products which conform to Open Group Product Standards
- Promoting the benefits of the IT DialTone to customers, vendors, and the public

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of Technical Standards (formerly CAE and Preliminary Specifications) through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product.

The ''X'' Device is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the Open Brand Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical Standards and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *Technical Standards* (formerly *CAE Specifications*)

  The Open Group Technical Standards form the basis for our Product Standards. These Standards are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a Technical Standard can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. Technical Standards are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *CAE Specifications*

  CAE Specifications and Developers' Specifications published prior to January 1998 have the same status as Technical Standards (see above).

- *Preliminary Specifications*

  Preliminary Specifications have usually addressed an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a Technical Standard. While the intent is to progress Preliminary Specifications to corresponding Technical Standards, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as Technical Standards, in which case the relevant Technology Specification is superseded by a Technical Standard.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the Technical Standards or Preliminary Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/corrigenda**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/pubs**.

This Document

This Backup Services API (XBSA) document is an Open Group Technical Standard. First, it presents an architecture for management and client applications which use the services of underlying backup and archive software. It then specifies a Backup Services API consisting of source procedure calls, type definitions and data structures, and return codes, to be used by these management and client applications.

The focus of this API is on defining backup services for data movement, as opposed to management, with an emphasis on retaining programming flexibility so as to ensure broad applicability. The programming model used aligns with the widely accepted open/read-write/close model.

Background information is included, which is intended to be of particular value to end users and vendors, covering the use of this API, and consideration of the wider issues involved in backup, archive, and restore operations.

Intended Audience

This Technical Standard is intended for three types of user:

1. Application developers who wish to write portable programs that use the backup service operations defined in this Technical Standard. Typical applications or end-user services would be databases, filesystems, document managers, and object managers.
2. Developers of management applications to manage the operation of backup services defined in this Technical Standard.
3. Developers of the actual backup services packages which carry out the functions of this Technical Standard. These services are provided to higher-level management applications or client programs.

Structure

The structure of this Technical Standard is as follows:

- Chapter 1 is an introduction to the scope and purpose of the XBSA API.
- Chapter 2 describes a conceptual architecture for backup and archive operations, which serves as a framework for XBSA.
- Chapter 3 gives an overview of the wider context and intended usage of the XBSA API, to support understanding of the services it defines.
- Chapter 4 gives the definitions for the XBSA function calls, in man-page format.
- Chapter 5 gives the type definitions and data structures used in XBSA.
- Appendix A considers issues of interest to developers of underlying backup services.
- Appendix B presents a sample C Language header file.

A Glossary and Index are also provided.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  — environment variables, which are also shown in capitals
  — utility names
  — external variables, such as *errno*
  — functions; these are shown as follows: *name*( ). Names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [ABCD] is used to identify a return value ABCD, including if this is an an error value.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items. In syntax the `|` symbol is used to separate alternatives, and ellipses (`...`) are used to show that additional arguments are optional.

Trademarks

Referenced Documents

The following referenced documents are relevant to this Technical Standard:

ISO POSIX-1
: ISO/IEC 9945-1:1996, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (identical to ANSI/IEEE Std 1003.1-1996). Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995 and 1003.1i-1995.

XSH, Issue 5
: CAE Specification, January 1997, System Interfaces and Headers, Issue 5 (ISBN: 1-85912-181-0, C606), published by The Open Group.

Chapter 1

Introduction

This document specifies the Open Systems Backup Services Data Movement Application Programming Interface (XBSA), which defines an interface between applications or facilities needing data storage management for backup or archive purposes, and the underlying services which provide these functions.

The programming interface defined in this document focuses on data movement. It is intended that data management interfaces will be covered in a separate document in the future. First, this document describes an overall architecture to serve as a basis for a general discussion of the XBSA functions. Then, in the context of this architecture, it specifies an API consisting of calling interfaces, structures and return codes. Appendices provide a glossary, additional information for end users and vendors on the use of this Technical Standard, and a sample C language header file.

XBSA resolves the need to standardize an API between users and applications needing backup services, and the underlying services available on many platforms. For example, file system and database vendors must integrate their products with popular backup products. Without the interfaces provided in XBSA, these vendors must negotiate individually with the backup products; with XBSA, they can use a consistent interface to interact with a variety of backup products.

1.1 Objectives

The architecture used in defining this Open Backup Services API addresses the following objectives. With XBSA, applications may be created to provide these functions:

- **Concurrent Services**
  A basic service to support both the backup/restoration and the archive/retrieval of tailored sets of data objects, including files, directories, and byte streams, for many different applications and users operating concurrently.
- **Object Searches**
  Facilities to support archive and backup object searches which enhance an end-user's chances, effort, and/or time to retrieve archived objects.
- **Scalability**
  The ability to accommodate a large volume of large data objects for long-term storage as well as small sets of data.
- **System Configurations**
  The ability to accommodate a wide spectrum of systems and system configurations, including:
  - — A single, standalone personal computer
  - — A client-server computing environment
  - — A widely distributed, heterogeneous (UNIX and non-UNIX) system consisting of many clusters of workstations and main-frame computers inter-connected through heterogeneous networks using different protocols

- **Integrity**
  A high level of integrity, including:
  — The consistency and atomicity of multiple operations
  — The ability to shield the effect of one set of operations from another set (transaction isolation)

The XBSA interfaces will support applications and services which meet the above objectives.

This API addresses portability issues across multiple backup products. It is not intended to address the interoperability of clients and backup servers across arbitrary networks; this is a responsibility of the underlying Backup Service product.

1.2 Terminology

Fundamental terms necessary to understand the architecture used in this Technical Standard are shown below. Further definitions are described in the Glossary.

XBSA Client
: Application-specific software which uses XBSA to request services from a Backup Service on behalf of a particular application. Typically an XBSA Client is tightly bound to a user application (such as a DBMS) or an operating system service (such as a file system) by existing in the address space of the application/service or being packaged with this function.

XBSA Manager
: Management software which uses XBSA to manage the services provided by a Backup Service. Typically an XBSA Manager may manage the operation of a variety of Backup Service implementations from a variety of vendors. For full functionality, an XBSA Manager will require additional interfaces which will be defined in a future Open Backup Services Management API Technical Standard. In the interim, management functionality will be achieved by implementation-specific mechanisms.

XBSA Application
: An XBSA Client or an XBSA Manager.

Backup Service
: Software which carries out the functions provided in XBSA. Independently-written packages implementing the lower XBSA interfaces may be written by various vendors for a variety of platforms. These packages provide interfaces meeting this Technical standard which may be dynamically or statically bound to an XBSA Application.

1.3 Overview

The portability of backup applications and related administration tools can be significantly improved with the standardization of the interfaces between these packages and an underlying backup service. Especially, if these interfaces can be managed as a system-independent, vendor-independent and application-independent API, vendors can focus on the more important aspects of their packages.

The approach adopted in this Open Backup Services API is therefore to define a minimal, generic and extensible API that is able to adequately and efficiently support a wide range of backup applications and utilities - from a simple backup utility for a standalone personal computer to a high-function, high-performance, multi-server solution for a heterogeneous distributed computing environment - while allowing the utmost flexibility in implementation and customization to meet specific needs.

In its simplest form, XBSA provides an interface between XBSA Applications and Backup Service implementations, as shown in Figure 1-1. The actual boundary between a particular XBSA Client and the related application or service is implemented by the developer of the application or service package.

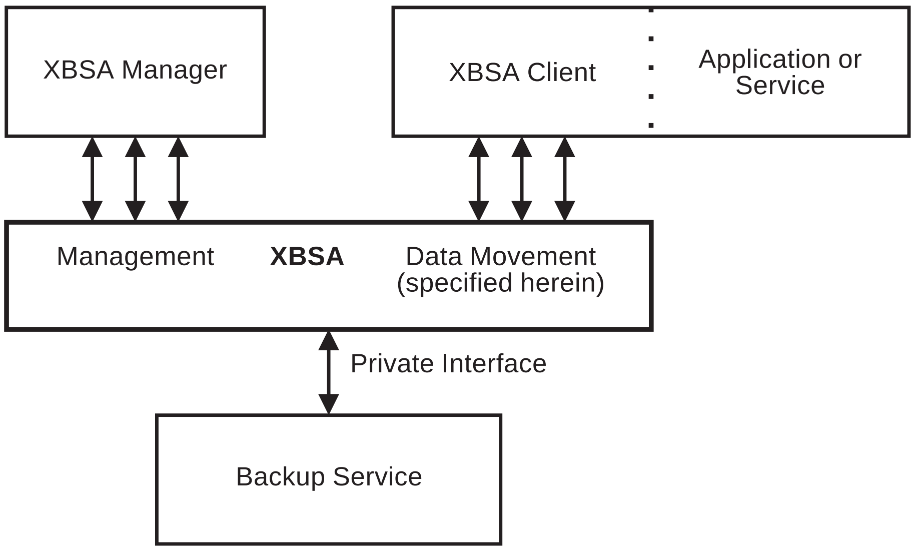

Figure 1-1 Simplified Architecture for XBSA

Figure 1-11 illustrates the data flow between the XBSA Application and the Backup Service. All operations are initiated by the XBSA Application.

This document defines the XBSA Data Movement API. It does not address the subject of a Management API for Backup Services.

Some of the parameter values used in XBSA calls are dependent on the specific Backup Service which is invoked. Consequently, while the XBSA calls and structures are generic in nature, an XBSA Application may need to determine the actual Backup Service being invoked in order to fully exploit all the features of the Backup Service.

Other aspects of a backup, archive, and restore (BAR) system, such as user interfaces, transfer protocols, media formats, and API(s) for specific environments are not covered by this Technical Standard. In the context of this Technical Standard, they are considered implementation options for a backup application or the underlying services.

In the same context, networking and remote support issues are not covered in this Technical Standard.

1.4 Deployment

By default, an implementation of the XBSA Backup Service will provide an object library using a standard name of the form **libxbsa.ext**, where **ext** is replaced by an operating system specific extension.

The XBSA interfaces may be deployed in either of two ways. The first is as a source-level interface, with an XBSA Application statically linking to the XBSA object library. The second option, which is expected to be more common, is as a binary-level interface, with the XBSA Application being dynamically linked to an XBSA object library.

The second method allows the XBSA Application the flexibility to bind to different Backup Services at run-time. The precise mechanism by which this achieved is dependent on the environment in which the XBSA Application is running. If the operating system provides support for a search path for locating dynamic libraries, this mechanism may be used to support multiple implementations simultaneously present on the same system. If this, or a similar mechanism, is not available, the XBSA Application will only be able to access a single Backup Service implementation using the default object library name and dynamic linking. In this case, multiple implementations may still be supported by means of static linking.

1.5 Conformance

An implementation claiming conformance to this Technical Standard must provide all the XBSA functions and data structures, as defined in Chapter 4 and Chapter 5.

1.6 Future Directions

The following items might be considered for potential future work activities.

- **Asynchronous Calls**
  Future extensions to this API may include asynchronous calls (particularly for tasks that require a long time for completion), calls to setup a separate data transfer path, and support for processing a set of objects in a single call.

- **Access Control**
  Future extensions may include enhanced security mechanisms, such as role-based user, delegation of authority, catalog access rights, user group and user agent as grantee, denial rules, audit trail, and mandatory control.

- **Extended Object Data Operations**
  Future extensions to the Open Backup Services API may include additional object data operations such as:

  — Retrieve substring, append, and replace

  — Additional object types to support different semantics

  — A mechanism to set up a separate data path to allow high-performance data transfer

  These extended capabilities are not needed to support a basic level of Backup Service.

- **Multi-party Commit Protocols**
  If any asynchronous update made by an XBSA Application is to be synchronized with XBSA operations, then the Open Backup Services API would need to be extended to include a multi-party commit protocol (for example, two-phase commit). Another possible extension of the API is a checkpoint/restart capability to support long transactions (for example, a full backup of a large-capacity medium or a group of media).

- **Management Functionality**
  Future work is anticipated to define management interfaces to provide services to XBSA Managers.

Chapter 2

Backup and Archive Architecture

This chapter presents a general architecture of a backup and restore environment to serve as a framework for discussing the Open Backup Services API functions in the next chapter.

In the course of describing this architecture, examples or scenarios will be given to illustrate the use of the XBSA interfaces. These examples are not intended to imply the only use of XBSA. The XBSA architecture was designed to permit a wide flexibility of use.

An overview of the general roles and responsibilities for backup and archive operations is provided in Section 2.1 on page 8, Section 2.2 on page 10, Section 2.3 on page 12, and Section 2.4 on page 13. The remaining sections of this chapter then introduce the terminology and descriptions of the XBSA model necessary to understand the API presented in the subsequent chapters.

2.1 Architectural Overview

Figure 2-1 shows a high-level view of this architecture, identifying the major components and highlighting the Open Backup Services API (XBSA). The functions provided by these components are described in turn below.

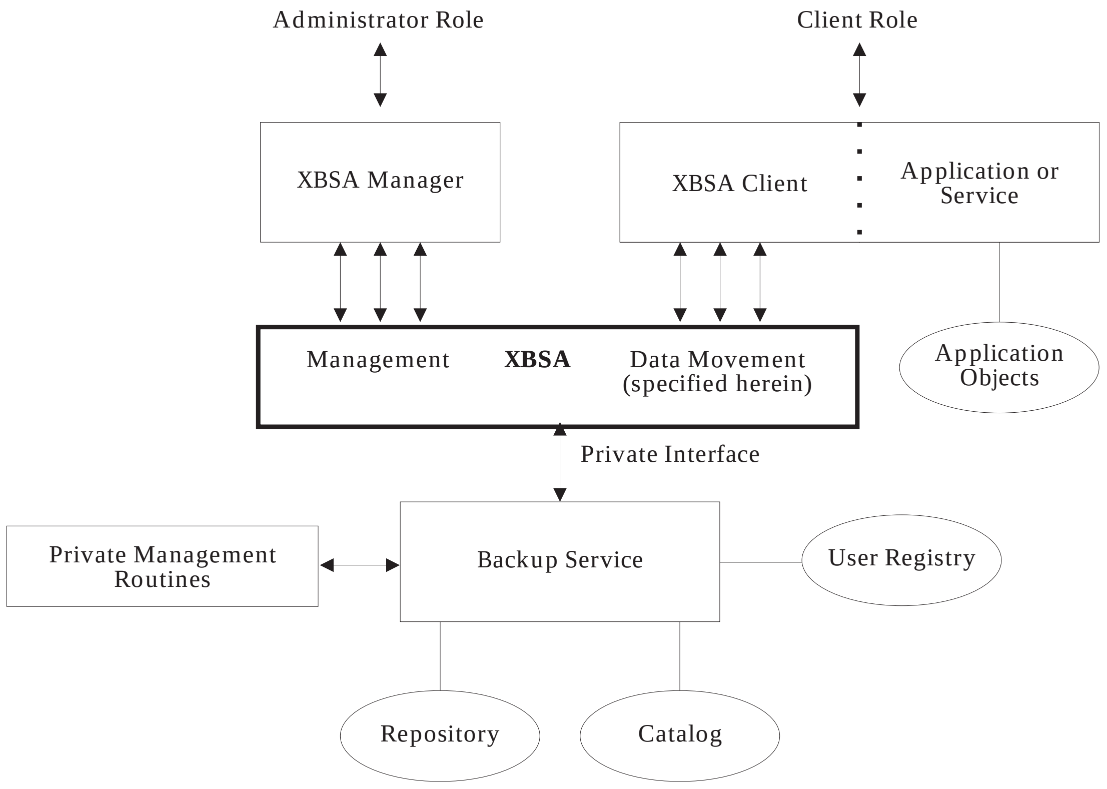

Figure 2-1 Context for XBSA Interfaces

Except for the specification of the XBSA Data Movement API, there are no internal or external interfaces specified for any of these components. For example, the user interfaces (**Administrator Role** or **Client Role**) are defined by the application developers, and the interfaces between a Backup Service and a particular catalog are left to the implementor of a particular Backup Service.

Design choices which pertain to trade-offs on performance, storage consumption or availability (reliability) are implementation decisions outside the scope of XBSA.

Any Backup Service implementation could have **Private Management Routines** (administrative interfaces) which are not visible to users through the Open Backup Services API. A specific Backup Service implementation may also have components not shown in Figure 2-1, such as an Index Manager, a Storage Manager, a Hierarchy Manager, a Log Manager, or a Communication Manager. Depending on the particular system configuration environment, certain system services may be used to support some of these components.

Furthermore, not indicated in Figure 2-1, a Backup Service implementation may be either local to a single system node or distributed across multiple system nodes.

The API Definitions defined in Chapter 4 comprise the XBSA Data Movement API. Through the use of these functions, the XBSA Application may use the data repository functions of a particular Backup Service.

2.2 Backup and Archive Operations

Through this API, one or more data objects may be backed up (that is, a backup copy of each object is created in the Repository controlled by the Backup Service). They can then be subsequently restored if the original objects become corrupted or lost due to storage or system failure, or to human error.

Similarly, one or more data objects may be archived (that is, an archive copy of each object is created in the Repository controlled by the Backup Service). This permits long-term, often low-usage, storage so as to reduce the storage cost. Machine-assisted management of these archived objects is facilitated, such that they can be subsequently retrieved when they are needed again.

Backup operations are frequently repetitive and tedious. Applications may provide a facility to support automated backup and to generate logs or reports. Thus XBSA expects that the XBSA Application (not the Backup Service) will actually initiate and control the data transfer functions.

Although Backup/Restore functions are operationally similar to Archive/Retrieve functions, they differ in their usage and requirements. This is primarily a user distinction relating to the way in which the XBSA API is used, rather than reflecting any specific features within the API itself.

The following discussion elaborates on these differences:

- **Semantics**
  Making a backup copy provides the ability to subsequently recover an active object or a set of active objects, for instance, in case of a system, network, or media failure, or a human error. Therefore, a backup copy could be created shortly after a new object is created or after an object is modified. This concept of backing up objects is application-independent.

  To reduce the amount of backup data, an incremental backup may be performed which backs up only objects which have been changed since a previous backup.

  An archive copy, on the other hand, provides long-term storage for an inactive object such as a file, which is often deleted from an application's repository after the archive copy is made. Therefore, an archive copy could be created when an object is classified as inactive, which is a policy of the application. For archives, there is no concept analogous to an incremental backup.

- **Retention**
  The useful life of a backup copy typically depends on the subsequently created backup copies for the same object. Usually a certain number of the most recent backup copies are retained, and the most recent copy might be retained as long as the original object remains active.

  The retention of an archive copy is usually determined by its age, or useful life, and does not depend on other copies of the same data object.

- **Availability**
  The availability and integrity of an archive copy should be assured once the copy is created, because the original application object may be (and often is) deleted. Therefore, archiving is logically a synchronous operation which may change the state of the original object; however, this is application dependent.

  A backup copy operation, on the other hand, may often be deferred in the interests of improving backup operation performance. This is acceptable because the original data object is still available to the application as the primary copy.

  For example, an automated backup operation may choose to skip an object which is being updated by a (possibly long-running) application. That object will then be backed up at the next scheduled backup time. To most applications, backing up an object (as opposed to keeping dual copies) is an asynchronous operation, which accepts the fact that there is an inherent window of inconsistency between a backup copy and the original object. An application or user needs to manage the risk level of this inconsistency in conjunction with performance and cost.

The application is responsible for implementing suitable availability policies using XBSA. By itself, XBSA does not provide availability.

- **Search**
  The retrieval of archive objects is usually selective and task-specific (application-wise). An archive object is often retrieved individually or in conjunction with a relatively small number of logically related objects. The ability to search through backup storage can be particularly useful to help a user to find the target objects to retrieve. XBSA supports queries of objects stored in the Backup Repository.

2.3 Open Backup Services API

Backup services are offered through a source-code Open Systems Backup Services API (XBSA) using a synchronous call interface. The structure of the calls is system-independent and application-independent.

The underlying part of the architecture that supports XBSA is called the **Backup Service**. The users of this API are collectively called **XBSA Applications**, which refers to both the dedicated management applications (**XBSA Manager**) and typical user applications which make use of the Backup Service (**XBSA Client**).

The end-users who backup/restore their data objects (**User Role**) or manage the Backup Service (**Administrator Role**) through XBSA use interfaces which are outside the scope of this Technical Standard.

At any point in time, a single Backup Service may support multiple XBSA Managers and multiple XBSA Clients, and a single XBSA Client that uses multiple processes can have concurrent calls through XBSA to the Backup Service.

Multiple Backup Service implementations can potentially exist on a single platform. Since the association of a particular XBSA Application to a particular Backup Service is dependent on which library of source code interfaces is used, the static or dynamic binding of executable modules determines which Backup Service is invoked. It should be noted that an XBSA Application in a single process can only interact with a single Backup Service at a time.

2.4 XBSA Applications

An XBSA Client makes use of certain Backup Services for specific applications or end-user services. Examples of these are a UNIX file system, a document management system, a database management system (DBMS), an information retrieval (IR) system, or a cache manager.

An XBSA Manager is also an end-user application, but it is distinguished in that its primary role is to provide management of services for an application through XBSA. This XBSA Manager may be provided by the vendor of the Backup Service, or an independent vendor may create a more universal manager. In either case, the Backup Service vendor could have private management interfaces (**Private Management Routines**) which are not defined in this Technical Standard.

An XBSA Client interacts with specific applications to access their **Application Objects** which they manage (for example, files, directories, byte streams). It then constructs suitable XBSA Objects from the application objects for backup / archive, calls the Open Backup Services API, reconstructs the application objects from restored / retrieved XBSA Objects, maintains a backup / archive log for the application it interacts with, engages a scheduler to automate deferred or periodic backup tasks, and possibly provides a customized end-user interface (graphical or command-line) to support interactive operations.

An XBSA Client also understands the requirements of the specific application, the format and relationships of the application objects, and the semantics of backup / archive (including incremental backup and consistency requirements). It may handle attribute extraction, data conversion, mappings of application objects to XBSA Objects (which do not have to be one-to-one), object packing, compression / decompression, encryption / decryption, constraint enforcement, and / or application-specific access control. It may itself support a distributed environment, such as interacting with a distributed file system or supporting a remote end-user interface via RPC calls.

The precise semantics of backup and restore for any application, and in particular the notion of **incremental backup**, is application-dependent and thus must be handled by a suitable XBSA Client. For example, the incremental backup of a file system directory (as a named set of files) may mean the backing up of the changes to the directory since the last backup was performed. These changes could include all the modified and newly created files in the directory, as well as all the file deletions.

On the other hand, an incremental backup of a large, continually appended log file may mean the backing up of all the appends added since the last backup. An incremental backup of a database may mean the backing up of only the modified pages in the database.

2.5 XBSA Session

An XBSA session is a logical connection between an XBSA Application and a Backup Service implementation. A session begins with a call to *BSAInit*( ) and ends with a call to *BSATerminate*( ). Nested sessions, and multiple sessions in the same address space, are not supported in this version of the Technical Standard.

Every XBSA session between a particular XBSA Application and Backup Service uses a **bsa_ObjectOwner**, which is registered with the Backup Service and used by Backup Service to control access to the objects it manages. This **bsa_ObjectOwner** is a fixed-length character string. No specific format is required (that is, it may be a combination of a node name and resource manager name, or a globally unique name associated with a distributed resource manager). Different instances of an XBSA Application may or may not use the same **bsa_ObjectOwner**. The **bsa_ObjectOwner** is authenticated for each session.

2.6 XBSA Objects and Object Names

This Open Backup Services API uses an object-based paradigm. Every data object visible and transferred at the XBSA interface is an **XBSA Object**, which is not to be confused with the **Application Object** shown in Figure 2-1 on page 8. The latter is an object managed by an application or service, and is the original source object which is to be backed up or archived.

2.6.1 Hierarchical Character Strings

A **hierarchical character string** has one or more components or fields, separated by a user-specifiable **delimiter character** (specified in the Backup Service-dependent Environment structure). Each component is a variable-length character string, and there is a length limit for the entire string. However, there is no restriction on the number of components (levels) a string can have. Two consecutive delimiters specify an empty level. A leading delimiter indicates an empty top level.

A hierarchical character string is interpreted one component at a time starting with the first component and ending whenever a resolution is made. For example, when comparing two strings, interpretation ceases whenever a distinction, or relative ordering, between the strings is determined. The delimiter character is implementation dependent and should be queried using *BSAGetEnvironment*( ).

Hierarchical strings facilitate the aggregation and segregation of objects, and may be used syntactically and structurally for object grouping, filtering, or selection. However, the semantics of a hierarchical name, including any naming constraints, is totally application-dependent, and unknown to the Backup Service.

A limited wild card capability is provided, see *BSAQueryObject*( ).

2.6.2 XBSA Object Naming

An XBSA Object has a two-part name: **BSA_ObjectOwner**, and **BSA_ObjectName**, which is summarized in Figure 2-2.

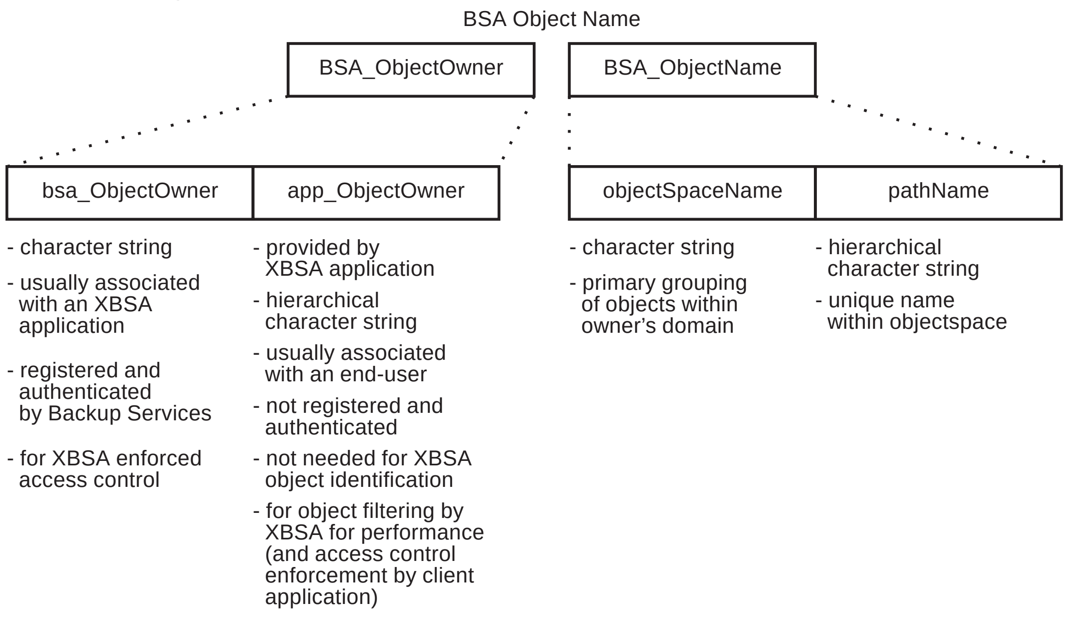
BSA Object Name
BSA_ObjectOwner
BSA_ObjectName
bsa_ObjectOwner
app_ObjectOwner
objectSpaceName
pathName
- character string
- usually associated with an XBSA application
- registered and authenticated by Backup Services
- for XBSA enforced access control
- provided by XBSA application
- hierarchical character string
- usually associated with an end-user
- not registered and authenticated
- not needed for XBSA object identification
- for object filtering by XBSA for performance (and access control enforcement by client application)
- character string
- primary grouping of objects within owner's domain
- hierarchical character string
- unique name within objectspace

Figure 2-2 Structure of an XBSA Object Name

The **BSA_ObjectOwner** is the name of the owner of the object and consists of two parts: **bsa_ObjectOwner**, and **app_ObjectOwner**. The **BSA_ObjectName** is the name assigned by the XBSA Application. It also consists of two parts: **objectSpaceName**, and **pathName**. The details of the data structures are defined in Chapter 5 on page 61.

As neither an **app_ObjectOwner** nor an **objectSpaceName** are required in the **BSA_ObjectDescriptor**, the **bsa_ObjectOwner** and **pathName** alone can name an XBSA Object. The **BSA_ObjectName** does not uniquely identify a single XBSA Object since multiple XBSA Objects with the same name can exist within the Backup Service (as multiple copies of the corresponding Application Object are made).

The Backup Service assigns to each XBSA Object a unique, persistent, fixed-length Object Identifier called **copyid** in the type definitions (see Chapter 5 on page 61), which remains unchanged throughout the life of the XBSA Object. This identifier must be used to retrieve an XBSA Object, subject to the applicable access control rules.

2.7 Security

Adequate security is a major requirement for any backup system. Authentication and access control may be administered via an underlying infrastructure. In those cases where it is not, a simple, but extensible, handle-based access model is supported by the Open Backup Services API, where the Backup Service provides session authentication and access control services for stored objects.

To support authentication, a registry of each authorized **bsa_ObjectOwner** and associated authorities may be maintained by the Backup Service. At the beginning of each XBSA session, the **bsa_ObjectOwner** may be verified against this registry, typically (though not necessarily) by means of a **BSA_SecurityToken** (for example, a password or a key). If the caller is authenticated, a unique handle is issued by the Backup Service to the XBSA Application for use during the session. The verification of the **bsa_ObjectOwner** may be performed internal to the Backup Service, or it may rely on the services of an external authentication system.

If the XBSA Client wants to alter this registry, communicate with the authentication system directly, or establish a new security token, the XBSA Client has to use a private API that has been provided by the Backup Service or by the authentication system. Such an API is not part of this Technical Standard.

The XBSA Application must then use the handle issued by the Backup Service in every subsequent API call during the session to identify itself as well as the particular session. This architecture allows an XBSA Application and a Backup Service to use a third-party authenticator (for example, Kerberos, which issues a similar validating token). The choice and operation of a particular validation system is implementation dependent.

An XBSA Application may also supply an **app_ObjectOwner** (for example, an end-user id) when initializing a session. If given, this name will not be authenticated by the Backup Service against the registry, but will be used for checking object ownership and access rules when objects are accessed during the session. If the **app_ObjectOwner** is not given, all objects owned by the **bsa_ObjectOwner** may be accessed.

Every XBSA Object is created by a **BSA_ObjectOwner**. As shown in Figure 2-2, a **BSA_ObjectOwner** consists of a **bsa_ObjectOwner** and **app_ObjectOwner** pair. The **bsa_ObjectOwner** possesses all access rights to the object.

The authentication of XBSA Application sessions as well as access control at that level are handled by the Backup Service. An XBSA Application, responsible for its own security domain, in turn handles the authentication and access control at the application level, possibly with the aid of the respective Application. The Backup Service provides object filtering support (utilizing the **app_ObjectOwner** provided by an XBSA Application to check object ownership and access rules) to reduce the amount of information that the XBSA Application has to retrieve, and thereby improving the overall retrieval performance.

2.8 Object Descriptors

An XBSA Object has a **BSA_ObjectDescriptor**, containing cataloging information and optional application specific object metadata. Cataloging information is capable of interpretation and searching by XBSA. Application-specific object metadata is not interpretable by XBSA but may be retrieved and interpreted by an application. Using an object's **objectName** or its assigned **copyId** identifier, the corresponding **BSA_ObjectDescriptor** and/or object data can be retrieved through the Open Backup Services API.

For the intended storage purpose (that is, backup or archive), each 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 XBSA Object has a **copyType** of either **backup** or **archive**, which is recognized by the Backup Service so that the two types of objects can be managed differently and accessed separately.

Furthermore, each XBSA Object has an **objectStatus** of either **most-recent** or **not-most-recent**. The significance of an XBSA Object's status is implementation defined. For example, an implementation might implement a policy that for backup copies, only the latest copy of an existing application object is active, whereas all archive copies are considered active.

In addition, to capture an application object's type information, the corresponding XBSA Object may have a **resourceType** (for example, "DOS filesystem") and a possibly resource-specific **BSA_ObjectType** (for example, **BSA_ObjectType_FILE**).

A **BSA_ObjectDescriptor** consists of a collection of object attributes. To be able to define an attribute, the appropriate attribute data type must be used in each XBSA Application and Backup Service implementation. 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, yyyymmddhhmmss.ss)

A particular Backup Service implementation may keep index(s) on some or all of these attributes to enhance search performance

The attributes are shown in the following table:

Attribute	**Data Type**	**Searchable?**
objectOwner (consisting of two parts) **bsa_ObjectOwner** **app_ObjectOwner**	 [fixed-length character string] [hierarchical character string]	yes
objectName (consisting of two parts) **objectSpaceName** **pathName**	 [fixed-length character string] [hierarchical character string]	yes
createTime	[date-time]	no
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

Table 2-1 Object Attributes

An XBSA Application may search for a particular XBSA Object within a certain search scope (for example, among objects belonging to an owner) by qualifying the search on the value of the appropriate searchable attributes. For character strings, a substring match (prefix, infix or suffix) is also supported.

On the other hand, non-searchable, application-specific attributes may be provided by an XBSA Application for storage in the **BSA_ObjectDescriptor**, but they are not interpreted by the Backup Service. They are stored in the XBSA Object attributes **objectInfo**, **resourceType**, and **objectDescription**.

Through this **objectInfo** attribute, application-specific metadata may be stored in the catalog so that this metadata can be efficiently retrieved without retrieving the actual object data stored in the repository.

Furthermore, this **objectInfo** attribute can also be used by an XBSA Application to maintain inter-object relationships and dependencies.

For more information on the **BSA_ObjectDescriptor** structure, see Section 5.4.3 on page 67.

2.9 Object Data

Object data contains the actual data entity that is archived or backed up by an XBSA Application. The Open Backup Services API supports only one type of object data, namely, a variable-length, unstructured and uninterpreted byte-string with a very large size limit.

To a particular XBSA Client, however, the XBSA Object Data can contain an internal structure that reflects the data of the Application Object or Objects that the XBSA Clients archived or backed up. In this context the XBSA Object Data can contain for example one of the following: a UNIX file system, a UNIX directory, a DOS file, a document, a disk image, a data stream, or a memory dump.

Through the Open Backup Services API, object data can be stored, retrieved, or deleted, but not searched or modified. Since object data may be stored on slow (or off-line) media, it is generally not advisable for an XBSA Application to store metadata in object data, especially information that could influence a data-retrieval decision.

However, the metadata of an XBSA Object which is stored in the catalog may be replicated in its object data to facilitate media interchange (interoperability). This is an XBSA Application implementation decision.

The mapping from the repository to physical storage media is a Backup Service implementation decision. A particular Backup Service implementation should accommodate different types of storage media, including disk, automated library, and shelf media. All opportunities for optimization or design trade-off are considered Backup Service implementation options. These include device exploitation, hierarchy management, object packing and placement, and redundancy.

2.10 Transaction Management

A backup task usually involves many updates to the catalog and repository. To protect their integrity and assure consistency between an Object Descriptor and the corresponding object data, and to provide atomicity for storing a group of objects, the concept of a transaction is used. Either all the operations within a **transaction** are correctly performed or none is performed.

Nested transactions are not permitted. One or more non-overlapping transactions may occur within any active session between an XBSA Application and a Backup Service. To define a transaction, an XBSA Application must **begin** a transaction before performing XBSA operations, and **end** the transaction after the appropriate XBSA calls have been made. Special XBSA calls are provided to begin and end transactions.The effect of a transaction is not visible until it is committed.

If a transaction is aborted by an XBSA Application, or if there is an error or failure encountered in the midst of a transaction, the transaction will be **rolled back** (that is, the effect of the transaction will be not be visible). It should be noted a rolled-back transaction may still have some observable effects. For instance, accumulated tape movement involved in the transaction may not be rewound. In such a case, no information relating to the data written to tape will be stored, and thus any information stored during the transaction will not be retrievable. In the case of a large, aborted transaction, considerable wastage of tape is possible.

The following XBSA calls are not affected by transaction management and take effect immediately without the possibility of being rolled back:

- *BSAQueryApiVersion*( )
- *BSAQueryServiceProvider*( )
- *BSAGetEnvironment*( )
- *BSAGetLastError*( )
- *BSAGetObject*( )
- *BSAQueryObject*( )
- *BSAGetNextQueryObject*( )

All XBSA backup and restore operations are required to be contained within transactions. Although XBSA restore operations are in a transaction, it does not provide any useful functionality to the XBSA Application. Transactions are constrained to be uni-directional, that is, it is not permitted to mix backup and restore operations in the same transaction.

Chapter 3

Overview of Backup Services API

The Open Backup Services API is a set of procedures which may be called up, available in a dynamic or static library provided by a vendor implementing a Backup Service. For the rest of this Technical Standard, the terms API or XBSA refer to this **Open Backup Services API**.

A synchronous interface in the C language is defined. If an application needs asynchronous behavior, it can be achieved by using its process or thread mechanisms.

The following sections in this chapter give a brief overview of the API calls and their intended usage. A detailed specification of the procedures comprising the API is given in the manual-page definitions in Chapter 4.

Chapter 5 defines the data type definitions and data structures used in the API.

3.1 Initialization and Authentication

It is necessary for an XBSA Application to set up a session with the Backup Service by invoking the *BSAInit*( ) call, in order to use these services. The procedures *BSAQueryApiVersion*( ) and *BSAQueryServiceProvider*( ) may be invoked prior to calling *BSAInit*( ) to determine the current version of the API used by the Backup Service and a string describing the provider of the Backup Service, respectively. The *BSAInit*( ) call authenticates the caller, sets up a session with the Backup Service and sets up an environment for the caller to be used in subsequent calls. A session set up by a *BSAInit*( ) call is terminated by a *BSATerminate*( ) call, which will release any resources acquired during the setting up of the session. Nested sessions, and concurrent sessions in the same address space are not permitted.

BSAInit( ) may authenticate the caller using a security token; the exact definition and use of this security token is implementation dependent. If a NULL security token is provided, the Backup Service can use a default, implementation-dependent security mechanism. In addition to those environmental parameters defined in this Technical Standard, there may also be additional, implementation dependent environmental parameters to the *BSAInit*( ) call.

A potential failure to authenticate the XBSA Client will cause the *BSAInit*( ) call to return BSA_RC_AUTHENTICATION_FAILURE. The XBSA Client will have to communicate with the security subsystem that is being used by the Backup Service to resolve this failure. The XBSA itself does not provide details about the reason of the authentication failure, or the means for correcting the authentication failure.

3.2 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 the calls 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:

- Object modification transactions - in which objects may be created or deleted.
- Object retrieval transactions - in which objects may only be queried 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 will result in a BSA_RC_INVALID_CALL_SEQUENCE error. The permissible call sequences are defined later in this chapter.

The following example illustrates the call pattern that may be used by a caller who, in one transaction, loops through a list of object names to backup each object.

```
if ((rc = BSAInit(bsaHandlePtr, tokenPtr,
                  userNamePtr, environmentPtr) ) != BSA_RC_SUCCESS) {
   <error processing>;
}

if ((rc = BSABeginTxn(bsaHandle) ) != BSA_RC_SUCCESS) {
    <error processing>;
}

<loop through all object names> {
    if ((rc = BSACreateObject(bsaHandle, objectDescriptorPtr,
                              dataBlkPtr)) != BSA_RC_SUCCESS) {
        <error processing>;
    }

    <loop through all data for one object> {
        if ((rc = BSASendData(bsaHandle,
                              dataBlkPtr) ) != BSA_RC_SUCCESS) {
            <error processing>;
        }

    } /* loop through all data for one object */

    if ((rc = BSAEndData(bsaHandle) ) != BSA_RC_SUCCESS) {
        <error processing>;
    }
} /* loop through all object names */

if ((rc = BSAEndTxn(bsaHandle,BSA_COMMIT) ) != BSA_RC_SUCCESS) {
    <error processing>;
}
```

```
if ((rc = BSATerminate(bsaHandle) ) != BSA_RC_SUCCESS) {
    <error processing>;
}
```

In this example, the data for each of the backup copy objects may be buffered by the API before sending it to the Backup Service.

3.3 Backup and Archive

An XBSA Application can create an XBSA object (either a backup copy or an archive copy) using the *BSACreateObject*( ) call, and pass the object's data in buffers using a sequence of *BSASendData*( ) calls ended by 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.

A scenario for creating a backup copy of an object is shown in Figure 3-1.

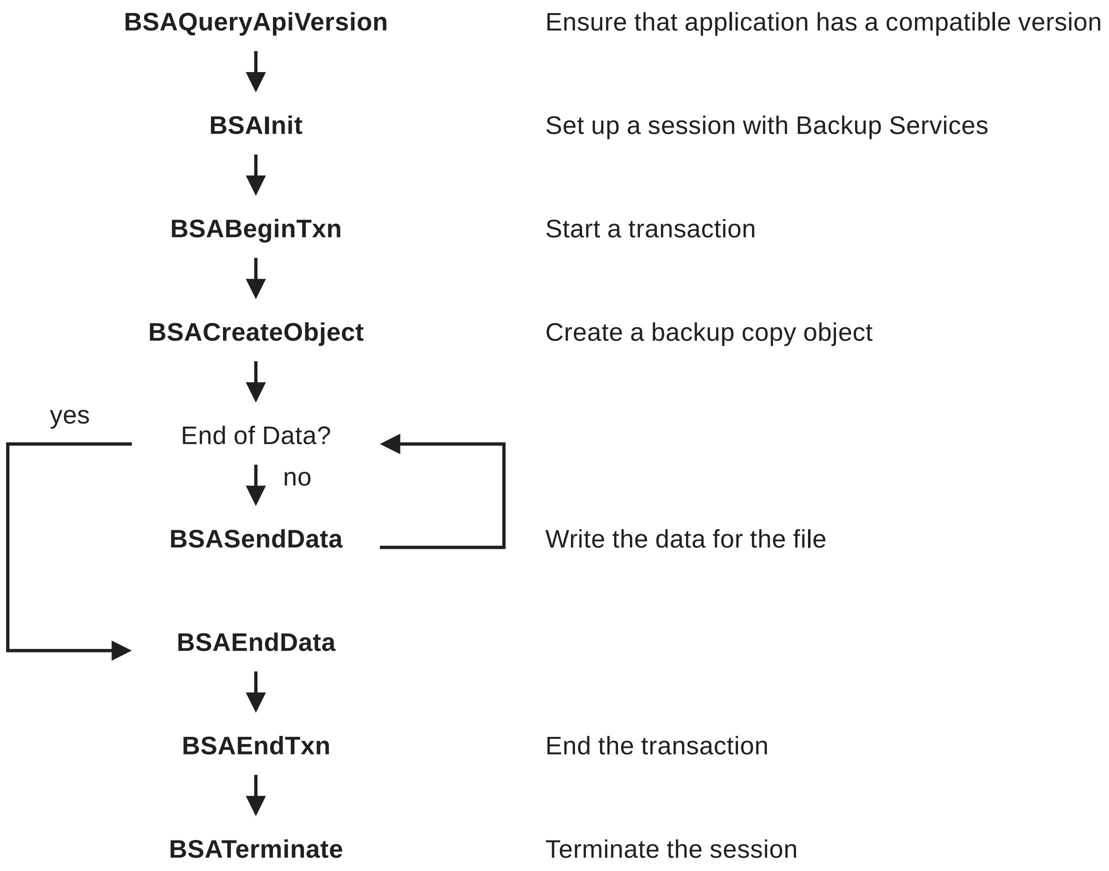

Figure 3-1 Creating a Backup Copy of an Object

3.4 Restore and Retrieve

The Restore and Retrieve interface is similar to the Backup and Archive interface, except that the data flow is reversed. The *BSAGetData*( ) call is used instead of the *BSASendData*( ) call. Data directed to standard output may be piped to a filter or command.

The *BSAGetObject*( ) call is used to restore (from a backup copy) or retrieve (from an archive copy) objects. The *BSAGetData*( ) call is used to get data for the object in buffers, and the *BSAEndData*( ) call is to signal the end of getting data for the object. A scenario for restoring an object is shown in Figure 3-2.

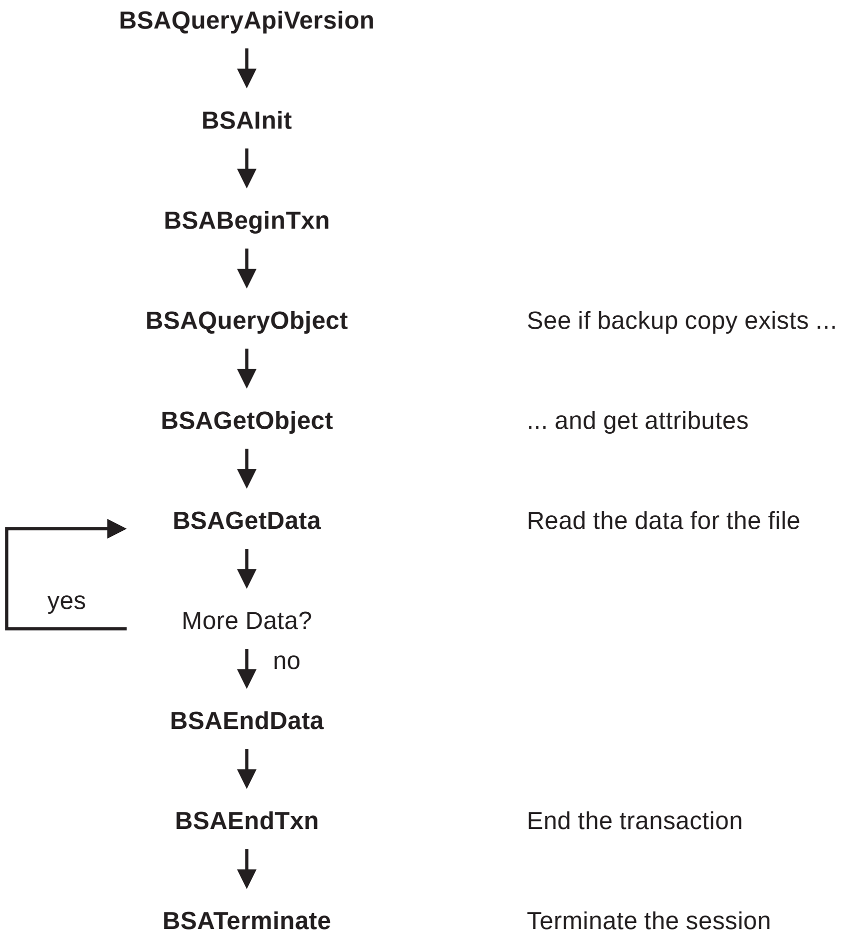

Figure 3-2 Restoring an Object

It should be noted that the use of transactions for restore and retrieval operations does not provide any functional benefit to the calling XBSA Application.

3.5 Query

An XBSA Application may query the Backup Service for XBSA Objects that have been created. The *BSAQueryObject*( ) call is used to query the Backup Service for objects. Since retention of objects is a function of the Backup Service's implementation there is no guarantee that the call to *BSAQueryObject*( ) will return an object descriptor.

The query is based on a subset of the Object Descriptor attributes, contained in a Query Descriptor. The result of a query can return Object Descriptors, but never XBSA Object Data. If the result is multiple Object Descriptors, the query result is retrieved one Object Descriptor at a time by using a succession of *BSAGetNextQueryObject*( ) calls.

The XBSA Application can retrieve the XBSA Object Data by using the *BSAGetObject*( ) call with the copyId from one of the object descriptors returned by the query.

For details of the **BSA_QueryDescriptor** structure, see Section 5.4.6 on page 70.

3.6 API Call Sequences

The permitted API call sequences are shown in the following diagrams. Any violation of these sequences will result in a bad call sequence error.

In the call sequence diagrams, the following conventions are used:

- States have arbitrary names.
- Function names have been shortened by dropping the "BSA" prefix.
- States from which *BSAEndTxn*(BSA_COMMIT) can be called successfully are indicated by a double box and named using **bold font**. Attempts to call *BSAEndTxn*(BSA_COMMIT) from any other state will return a BSA_RC_TRANSACTION_ABORTED error.

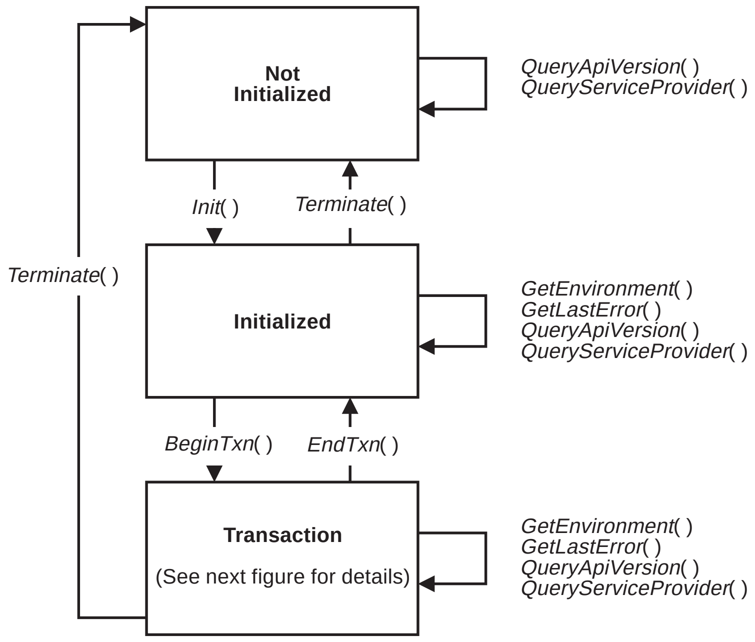

Figure 3-3 Permitted Call Sequences - Overview

Leaving the "Transaction" state with other than a successful call of *BSAEndTxn*(BSA_COMMIT) will cause the transaction to be aborted. If the transaction was creating or deleting an object, that object will not be created or deleted. Apart from this, a transaction does not have any other side effect. In particular, the state of the storage media is not restored to the state just before the start of the aborted transaction.

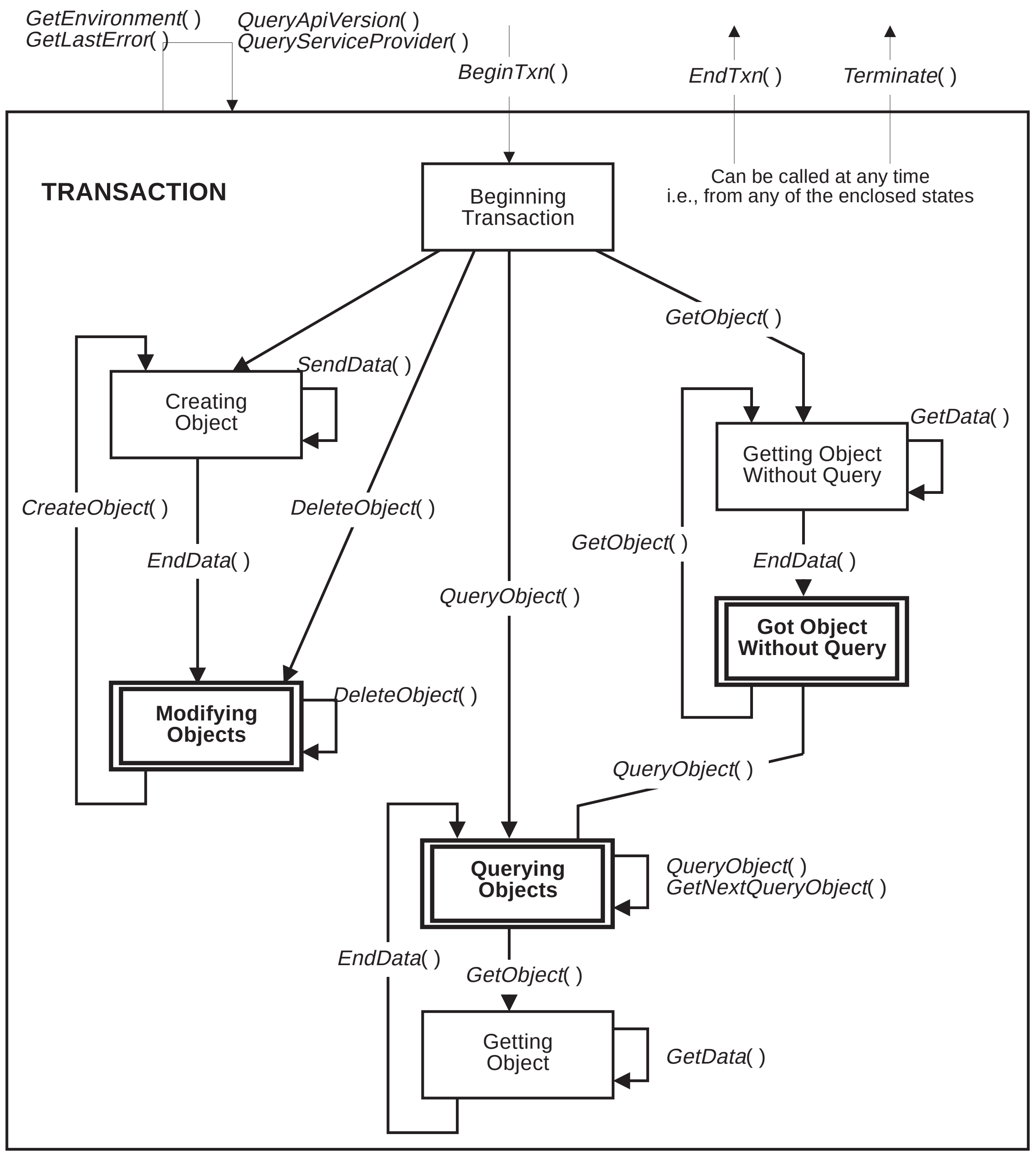

Figure 3-4 Permitted Call Sequences - Transactions

Since getting or querying an object does not alter the state of the object, leaving the transaction has no side effects.

3.7 Buffers

All buffers that are used by the XBSA are allocated by the XBSA Application. The Backup Service fills data into the buffers, but never allocates any memory that is passed back to the XBSA Application. This simplifies buffer allocation and deletion since the XBSA Application is solely responsible.

However, to allow the Backup Service to influence how buffers should be allocated, and to provide the Backup Service with the ability to reserve private sections in certain buffers, several conventions are used in the XBSA.

3.7.1 Buffer Size

For function calls that specify the size of the buffer as a separate parameter, XBSA uses the following convention to allow the Backup Service to signal that a buffer is not large enough and provide the XBSA Application with the means to discover what the correct size should be.

The parameter that specifies the size is a pointer, so that the Backup Service can alter the parameter. The size is always in bytes. If the size is adequate and a valid buffer is given, the Backup Service will copy the requested data into the buffer and set the actual size in the size parameter.

If the size is inadequate, the Backup Service will not copy the data into the buffer. It will set the size parameter to the actual size of the data to be copied and return 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*( ) and *BSAQueryServiceProvider*( ).

3.7.2 Private Buffer Space

For function calls that use the **BSA_DataBlock32** structure, a convention has been adopted that allows the Backup Service to reserve certain portions of the buffer for its own use. There are two areas that can be reserved by the Backup Service:

- 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 reserved for the XBSA Application is the:

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

To make this preference known to the XBSA Application, the Backup Service sets 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 not used for passing data but for passing the Backup Service's preference. The parameters that must be set by the Backup Service, and their meaning, are given in Table 3-1.

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

Table 3-1 Parameters in the BSA_DataBlock32 Structure

Subsequent calls to *BSAGetData*( ) or *BSASendData*( ) must adhere to the preferences that were specified by the Backup Service. The relationship between the areas, and their mapping to the fields of the **BSA_DataBlock32** structure is described in Section 5.4.2 on page 65.

The Backup Service can write anything into the header and trailer area of the actual buffer, as specified by the **bufferPtr** parameter in the **BSA_DataBlock32** structure.

3.7.3 Use of BSA_DataBlock32 in *BSASendData*()

For *BSASendData*( ), all parameters in the **BSA_DataBlock32** structure must be set by the XBSA Application and adhere to the Backup Service preferences or the function will fail with a BSA_RC_INVALID_DATABLOCK error. The Backup Service is not allowed to change any of the parameters.

3.7.4 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 Backup Service preferences or the function will fail with a BSA_RC_INVALID_DATABLOCK error. The Backup Service must change the following parameter:

- **numBytes**
  Set the actual number of bytes copied into the data segment.

The Backup Service is not allowed to change any of the other parameters.

3.7.5 Shared Memory

The **BSA_DataBlock32** structure contains fields that allow the use of shared memory blocks for passing data between an XBSA Application and a Backup Service.

The **shareId** specifies an identifier that can be used by the Backup Service to map the memory in which the buffer resides into its address space. A special typedef, **BSA_ShareId**, is used to define the *shareId* field using a type appropriate for the operating environment. For UNIX systems, this is based on the shared memory semantics defined in the Single UNIX Specification, System Interfaces and Headers, Issue 5 (see reference **XSH, Issue 5**) and the **shareId** is the *shmid* identifier obtained by calling the *shmget*( ) function. For NT systems, it is a file handle which will allow the Backup Service to map the file in memory.

The **shareOffset** specifies where in shared memory the buffer starts.

The semantics for the **shareId** are given in Table 3-2.

shareId == -1	This buffer is not available in shared memory. The **shareOffset** field is undefined. The **bufferPtr** is a true pointer in the address space of the client.
shareId != -1	The Backup Service may map this buffer into its memory by using the appropriate operating system services. The **shareOffset** becomes an offset into the shared memory. The *bufferPtr* is a true pointer in the address space of the client. Note that the Backup Service is not forced to map the buffer into its address space. It can copy the buffer in the address space of the XBSA Client.

Table 3-2 Semantics for **shareId**

Since the client may be using multiple buffers, it is not guaranteed that only one **shareId** is used in the object data transfers. It is guaranteed, however, that a **shareId** itself will not change during a session. In other words, the XBSA Client is not allowed to re-map a **shareId** within a session once it has been made available to the Backup Service. In this way the Backup Service can keep a simple lookup table to see whether the **shareId** is currently mapped in its address space.

The following two sections describe how shared memory buffers are used in the UNIX and Microsoft Windows NT operating systems.

3.7.5.1 Shared Memory on UNIX

For UNIX, the scenario for shared memory is as follows:

- The client performs either a *BSACreateObject*( ) or *BSAGetObject*( ) call to start the transfer and receives the buffer parameters.
- The client allocates a buffer in shared memory, thereby receiving a shared memory identifier. The client fills the buffer with data, if necessary.
- The client calls *BSASendData*( ) or *BSAGetData*( ) with the buffer parameters and the identifier of the shared memory block it previously allocated.
- The library implementation forwards the information to a different process, which will have to map the shared memory into its address space. This process can use *shmctl*( ) to retrieve the size of the shared memory segment, and use *shmat*( ) to attach the memory to its address

space. Once attached, it uses the **shareOffset** to locate the actual buffer, and from there the normal **BSA_DataBlock32** rules apply.

3.7.5.2 Shared Memory on NT

For NT, the scenario for shared memory is as follows:

- The client performs either a *BSACreateObject*( ) or *BSAGetObject*( ) call to start the transfer and receives the buffer parameters.
- The client allocates a buffer in shared memory, thereby receiving a file handle. The shared memory is set up by creating a file-mapping object using *CreateFileMapping*( ), and mapping it the client's address space with *MapViewOfFile*( ). The client fills the buffer with data, if necessary.
- The client calls *BSASendData*( ) or *BSAGetData*( ) with the buffer parameters and the file handle of the file it used for mapping the shared memory.
- The library implementation forwards the information, including the process ID of the client, to a different process. This process will have to duplicate the handle using *OpenProcess*( ) and *DuplicateHandle*( ), and map the file in its address space with *MapViewOfFile*( ). Once mapped, it uses the **shareOffset** to locate the actual buffer, and from there the normal **BSA_DataBlock32** rules apply.

For details of the interfaces, refer to the Microsoft Windows NT documentation.

Chapter 4

Backup Services API Definitions

4.1 General

This Open Backup Services (XBSA) document specifies an Applications Programming Interface (API), which provides an interface between applications or facilities needing data storage management for backup or archive purposes, and the underlying services which provide these functions.

The approach adopted in XBSA is to define a minimal, generic API that is able to adequately and efficiently support a wide range of backup applications and utilities. The API addresses portability issues across multiple backup products. It is not intended to address the interoperability of clients and backup servers across arbitrary networks; this is a responsibility of the underlying Backup Service product.

The following man pages describe the function calls used in the XBSA API. Many of the parameter values used in the XBSA calls are dependent on the specific Backup Service invoked from a particular vendor. Consequently, while some XBSA calls and structures are generic, any particular XBSA Application may need to determine the actual Backup Service being invoked in order to correctly use XBSA.

This chapter contains the C language definitions for all the Open Backup Services API procedures. Chapter 5 gives the data type definitions used in the API.

In the descriptions in these man-pages:

(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.

4.2 Return Code Values

The following XBSA Return Code Values table lists the possible return codes for the XBSA function calls. Each individual manual page lists precisely which return codes are valid for each interface.

The return code [BSA_RC_SUCCESS] is returned on successful completion by all XBSA interfaces.

Return Code	Value	Meaning
[BSA_RC_ABORT_SYSTEM_ERROR]	0x03	System detected error, operation aborted.
[BSA_RC_ACCESS_FAILURE]	0x4D	Access to the requested object is not possible.
[BSA_RC_AUTHENTICATION_FAILURE]	0x04	There was an authentication failure.
[BSA_RC_BUFFER_TOO_SMALL]	0x4E	The supplied buffer is too small to contain the data, as specified by the accompanying size parameter.
[BSA_RC_INVALID_CALL_SEQUENCE]	0x05	The sequence of API calls is incorrect.
[BSA_RC_INVALID_COPYID]	0x4F	The **copyId** field contained an unrecognized value.
[BSA_RC_INVALID_DATABLOCK]	0x34	The **BSA_DataBlock32** parameter contained an inconsistent value.
[BSA_RC_INVALID_ENV]	0x50	An entry in the environment structure is invalid or missing.
[BSA_RC_INVALID_HANDLE]	0x06	The handle used to associate this call with a previous *BSAInit*( ) call is invalid.
[BSA_RC_INVALID_OBJECTDESCRIPTOR]	0x51	The **BSA_ObjectDescriptor** was invalid.
[BSA_RC_INVALID_QUERYDESCRIPTOR]	0x53	The **BSA_QueryDescriptor** was invalid.
[BSA_RC_INVALID_VOTE]	0x0B	The value specified for **vote** is invalid.
[BSA_RC_NO_MATCH]	0x11	No object matched the specified predicate.
[BSA_RC_NO_MORE_DATA]	0x12	No more data is available.
[BSA_RC_NULL_ARGUMENT]	0x55	A NULL pointer was encounterered in one of the arguments
[BSA_RC_OBJECT_NOT_FOUND]	0x1A	There is no copy of the requested object.
[BSA_RC_SUCCESS]	0x00	The function succeeded.
[BSA_RC_TRANSACTION_ABORTED]	0x20	The transaction was aborted.
[BSA_RC_VERSION_NOT_SUPPORTED]	0x4B	The Backup Service implementation does not support the specified version of the interface.

Table 4-1 XBSA Return Code Values

4.3 XBSA Function Definitions

The Backup Services function definitions are presented in the remainder of this Chapter.

Function Call	Operation
BSABeginTxn( )	begin an API transaction
BSACreateObject( )	create a BSA object (either a backup or an archive copy)
BSADeleteObject( )	delete a BSA object
BSAEndData( )	end a *BSAGetData*( ) or *BSASendData*( ) sequence
BSAEndTxn( )	end a transaction
BSAGetData( )	get a byte stream of data using buffers
BSAGetEnvironment( )	retrieve the current environment for the session
BSAGetLastError( )	retrieve the error code for the last system error
BSAGetNextQueryObject( )	get the next object relating to a previous query
BSAGetObject( )	get an object
BSAInit( )	initialize the environment and set up a session
BSAQueryApiVersion( )	query for the current version of the API
BSAQueryObject( )	query about object copies
BSAQueryServiceProvider( )	query the name of the Backup Service implementation
BSASendData( )	send a byte stream of data in a buffer
BSATerminate( )	terminate a session

Table 4-2 XBSA Function Calls

NAME

BSABeginTxn — begin an API transaction

SYNOPSIS

```
#include <xbsa.h>

int BSABeginTxn(long bsaHandle)
```

DESCRIPTION

The *BSABeginTxn*( ) call indicates to Backup Service the beginning of one or more actions that will be executed as an atomic unit, that is, all the actions will succeed or none will succeed. An action can be assumed to be either a single API call or a series of API 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 will be aborted.

Nested transactions are not supported.

PARAMETERS

long *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.

[BSA_RC_INVALID_HANDLE]
The handle used to associate this call with a previous *BSAInit*( ) call is invalid.

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSACreateObject — create a BSA object (either a backup or an archive copy)

SYNOPSIS

```
#include <xbsa.h>

int BSACreateObject(long bsaHandle,
                    BSA_ObjectDescriptor *objectDescriptorPtr,
                    BSA_DataBlock32 *dataBlockPtr)
```

DESCRIPTION

The *BSACreateObject*( ) call creates an an object within the Backup Service. All objects created are owned by the **bsa_ObjectOwner** creating the object. For backup and archive copy objects, duplicate BSA_ObjectNames are allowed.

The *BSACreateObject*( ) call is used to create an object when the object's data is passed in memory buffers. The dataBlockPtr parameter in the *BSACreateObject*( ) call allows the caller to obtain information about the buffer structure required by the Backup Service.

The object's 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 object. The *BSASendData*( ) and *BSAEndData*( ) calls must follow the *BSACreateObject*( ) call and must be in the same transaction.

PARAMETERS

long *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 object attributes, including its name, copy type, and so on.

BSA_DataBlock32 **dataBlockPtr* (O)
This parameter points to a structure that is used to obtain the details of the required buffer structure.

EXTENDED DESCRIPTION

Within the 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 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 will default to the value specified in *BSAInit*( ).

The following fields in the 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 Client intends to create a non-empty XBSA Object (that is, there will be XBSA Object Data). This size is in bytes. Although the actual XBSA Object Data may be more or less than the estimate, the Backup Service may return a failure if the XBSA Object Data is orders of magnitude larger than the **estimatedSize**. If the **estimatedSize** is zero, this call must be followed by an *BSAEndData*( ) without calling *BSASendData*( ) in between. The **estimatedSize** can be used by the Backup Service as a hint, for example to select the storage medium or storage device.

The Backup Service may return several values to the BSA Application through the **objectDescriptorPtr** for a newly created XBSA Object. The Backup Service returns either all or none of these values.

The **copyId** attribute is a persistent, fixed-length Object Identifier which remains unchanged throughout the life of the object.

It is the responsibility of the Backup Service to assign **copyId** values to objects. When objects are deleted, their copyIds are returned to the pool of available identifiers, but the Backup Service must choose an assignment policy that prevents their immediate reuse.

If the **copyId** field is non-zero, the Backup Service returned values for the **copyId**, **createTime**, **restoreOrder**, and **objectStatus** fields. If the **copyId** field is zero, the values for the **createTime**, **restoreOrder**, and **objectStatus** fields are undefined. The calling BSA Application must call *BSAQueryObject*( ) to obtain an updated object descriptor.

Note: If the returned **copyId** is zero and the XBSA Client uses *BSAQueryObject*( ) to retrieve an updated BSA_ObjectDescriptor for the newly created object, the Backup Service cannot guarantee that the **ObjectDesriptor** returned by *BSAQueryObject*( ) is for the object created by this call to *BSACreateObject*( ).

The behavior of returning a **copyId** value of zero is permitted for backwards compatibility with existing implementations. This behavior is deprecated and it is not intended that it should be used in new implementations.

The **createTime** field is in UTC. The **restoreOrder** field is optional and can have the value zero, which means that the Backup Service did not specify a restore order.

The **dataBlockPtr** structure does not point to an actual buffer. All values in the **dataBlockptr** should be zero, and will be overwritten. The structure is used by the Backup Service to provide the BSA Application with the Backup Service's preference for the structure of the data blocks that will be used to pass the object's data. The BSA Application should examine the values returned in order to determine the buffer structure that it should create. The significance of the returned values is as follows:

bufferLen == 0	The Backup Service has no restrictions on the buffer length. No trailer portion is required.
bufferLen != 0	The Backup Service 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	The Backup Service 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 the Backup Service must not exceed **numBytes** bytes.
headerBytes == 0	The Backup Service only accepts buffers with no header portion.
headerBytes != 0	The length of the header portion of buffers accepted by the Backup Service is **headerBytes** bytes.
bufferPtr	Not used

The values returned by the Backup Service must conform to the relationships defined in Section 5.4.2 on page 65.

The values returned by the call to *BSACreateObject*( ) remain in effect for the duration of the data transfer of the object being created, that is, until the next *BSAEndData*( ) call.

It is the responsibility of the XBSA Client to provide this guarantee, if so desired. For example, the XBSA Client could use a unique identifier in the **objectDescription** field of the BSA_ObjectDescriptor for such a purpose.

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 object with 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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSADeleteObject — delete a BSA object

SYNOPSIS

```
#include <xbsa.h>

int BSADeleteObject(long bsaHandle, BSA_UInt64 copyId)
```

DESCRIPTION

The *BSADeleteObject*( ) call deletes an XBSA object from the Backup Service. The value for copyId can be obtained from a previous *BSAQueryObject*( ) call. The copyId value is unique within the Backup Service. A **bsa_ObjectOwner** can only delete objects that it owns.

It is not possible to create and then delete the same object within a single transaction.

PARAMETERS

long *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 object to be deleted. The value(s) for a specific 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 object with 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.

NAME

BSAEndData — end a *BSAGetData*( ) or *BSASendData*( ) sequence

SYNOPSIS

```
#include <xbsa.h>

int BSAEndData(long 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*( ) will not transfer any more data for the object to the caller. When used with *BSACreateObject*( ) or *BSASendData*( ) calls, *BSAEndData*( ) tells the Backup Service that the caller has finished sending data for a particular 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

long *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.

NAME

BSAEndTxn — end a transaction

SYNOPSIS

```
#include <xbsa.h>

int BSAEndTxn(long 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 can only be returned when a vote of BSA_Vote_COMMIT has been specified but an error has occurred which causes the transaction to be aborted.

PARAMETERS

long *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.

NAME

BSAGetData — get a byte stream of data using buffers

SYNOPSIS

```
#include <xbsa.h>

int BSAGetData(long bsaHandle, BSA_DataBlock32 *dataBlockPtr)
```

DESCRIPTION

BSAGetData( ) allows the caller to request a buffer full of XBSA Object Data from the Backup Service. This call is used after a *BSAGetObject*( ) call or after other *BSAGetData*( ) calls.

PARAMETERS

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

BSA_DataBlock32 **dataBlockPtr* (I / O)
This parameter points 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 will return, in this structure, the number of bytes of data that have been sent to the caller for this call.

EXTENDED DESCRIPTION

The Backup Service overwrites the **numBytes** field to provide the actual values used. The Backup Service may not modify any other fields. The BSA Application may only use the data portion of the buffer, in which the 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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSAGetEnvironment — retrieve the current environment for the session

SYNOPSIS

```
#include <xbsa.h>

int BSAGetEnvironment(long bsaHandle, BSA_UInt32 *sizePtr,
                      char **environmentPtr)
```

DESCRIPTION

The *BSAGetEnvironment*( ) call returns the (keyword, value) pairs that are currently defined in the environment for the session. This call is used to retrieve environment variables from the Backup Service environment.

PARAMETERS

long *bsaHandle* (I)

This parameter is the handle that associates this call with a previous *BSAInit*( ) call. It identifies the session.

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 "=" followed by a null-terminated value. The array of pointers is terminated by a NULL pointer.

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, this will force a buffer too small error, thus providing a mechanism to query the required size.

The following environment variables, to be returned by the Backup Service implementation, are defined as part of this Technical Standard.

Variable Name	**Description**	**Format**
BSA_DELIMITER	The delimiter used in hierarchical character strings (default "/")	A single character drawn from the POSIX.1 portable filename character set for example, ":"
BSA_SERVICE_PROVIDER	Identifies the BSA implementation. This is the same string that is returned by the *BSAQueryServiceProvider*( ) function call.	A hierarchical character string with at least 3 fields as follows: CompanyName / ProductName / ProductVersion[/...] Additional, delimiter-separated, implementation-specific fields are permitted for example, IBM / ADSM / 2.1.3

Additional private variables are allowed. The *BSAGetEnvironment*( ) call only returns environment variables that are meaningful to the Backup Service. This allows the XBSA Application to discover which variables that it specified when it called *BSAInit*( ) were interpreted by the Backup Service.

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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

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 Backup Service implementation. It is used to return platform-specific information describing 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 points to a data area that contains a text string describing the last error encountered.

EXTENDED DESCRIPTION

If the Backup Service sets the *sizePtr* parameter to zero, the Backup Service is unable to return a string describing the last error. This indicates that the Backup Service 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 will force a [BSA_RC_BUFFER_TOO_SMALL] error, thus providing 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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSAGetNextQueryObject — get the next object relating to a previous query

SYNOPSIS

```
#include <xbsa.h>

int BSAGetNextQueryObject(long bsaHandle,
                          BSA_ObjectDescriptor *objectDescriptorPtr)
```

DESCRIPTION

The *BSAGetNextQueryObject*( ) call returns the next object descriptor in response to a previous query. Successive calls to *BSAGetNextQueryObject*( ) will return these object descriptors.

PARAMETERS

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

BSA_ObjectDescriptor **objectDescriptorPtr* (O)
This parameter points to an object descriptor structure that will be filled with the values for each object returned in turn.

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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSAGetObject — get an object

SYNOPSIS

```
#include <xbsa.h>

int BSAGetObject(long bsaHandle,
                 BSA_ObjectDescriptor *objectDescriptorPtr,
                 BSA_DataBlock32 *dataBlockPtr)
```

DESCRIPTION

BSAGetObject( ) retrieves the **BSA_ObjectDescriptor** for the XBSA Object identified by the **copyId** and prepares the Backup Service to retrieve the XBSA Object Data.

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

PARAMETERS

long *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 XBSA Object's **copyId** to the Backup Service.

BSA_DataBlock32 **dataBlockPtr* (O)
This parameter points 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 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 will be overwritten. The structure is used by the Backup Service to provide the XBSA Application with the Service's preference for the structure of the data blocks that will be used to pass the object's data. The XBSA Application should examine the values returned in order to determine the buffer structure that it should create. The significance of the returned values is as follows:

bufferLen == 0	The Backup Service has no restrictions on the buffer length. No trailer portion is required.
bufferLen != 0	The Backup Service 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	The Backup Service 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 the Backup Service must be **numBytes** bytes. If the Backup Service provides a larger data portion, the Backup Service may take advantage of it.
headerBytes == 0	The Backup Service only accepts buffers with no header portion.
headerBytes != 0	The length of the header portion of buffers accepted by the Backup Service is **headerBytes** bytes.
bufferPtr	Not used

The values returned by the Backup Service must conform to the relationships defined in Section 5.4.2 on page 65.

The values returned by the call to *BSAGetObject*( ) remain in effect for the duration of the data transfer of the object being created, that is, until the next *BSAEndData*( ) 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]
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]
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 encounterered in one of the arguments

[BSA_RC_OBJECT_NOT_FOUND]
The given **copyId** does not exist.

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSAInit — initialize the environment and set up a session

SYNOPSIS

```
#include <xbsa.h>

int BSAInit(long *bsaHandlePtr, BSA_SecurityToken *tokenPtr,
            BSA_ObjectOwner *objectOwnerPtr, char **environmentPtr)
```

DESCRIPTION

The *BSAInit*( ) call authenticates the XBSA Application, sets up a session with the Backup Service and an environment for subsequent API calls for the caller. Nested sessions, and concurrent sessions in the same address space are not supported.

PARAMETERS

long **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)

If a Backup Service implementation provides its own authentication and access control, this parameter points to a security token that is to be used to authenticate the XBSA Application. The authentication is valid for the session and all calls using the returned session handle will be assumed to be made by the same XBSA Application.

If a NULL security token pointer is provided, the Backup Service can use a default, implementation-dependent security mechanism.

If this call returns [BSA_RC_AUTHENTICATION_FAILURE] the XBSA Application should consult an implementation dependent security API to resolve the failure. The XBSA Application can use *BSAQueryServiceProvider*( ) to determine whether and what security API to use.

BSA_ObjectOwner **objectOwnerPtr* (I)

This parameter points to a structure used to specify both the **bsa_ObjectOwner** and the **app_ObjectOwner**. Only the **bsa_ObjectOwner** field is mandatory and must be specified using a non-empty null-terminated string. 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. If the **BSA_ObjectOwner.bsa_ObjectOwner** field for any of the XBSA calls used in this session is empty, it will default to the value specified in this call.

char ***environmentPtr* (I)

This parameter points 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. Each string consists of a keyword followed by an "=" and followed by a null-terminated value. The array of pointers is terminated by a NULL pointer.

EXTENDED DESCRIPTION

The following environment variables, to be provided by the XBSA Client, are defined as part of this Technical Standard.

Variable Name	Description	Format
BSA_API_VERSION	Mandatory. Specifies the version of the specification that the calling XBSA Application requires.	A string containing 3 numeric elements, (version, issue, level) separated by periods. For example, "1.1.0", identifies this document. See *BSAQueryAPIVersion*(). for example, 1.1.3
BSA_SERVICE_HOST	Optional. Identifies a specific host system on which the Backup Service is running. If this variable is not provided, an implementation-defined default host will be selected.	A string containing an implementation-defined host name. (Normally, this will depend on the underlying operating system environment.) for example, backup.xyz.com

Additional private variables are allowed as long as they are agreed upon by the XBSA Application and Backup Service. Variables defined by the XBSA Application but not interpreted by the Backup Service are silently ignored and not added to the Backup Service's environment variables. Variables required by the Backup Service and not specified by the client may result in a [BSA_RC_INVALID_ENV] error during a *BSAInit*( ) call. The *BSAGetEnvironment*( ) call only returns environment variables that are meaningful to the Backup Service. This allows the XBSA Application to discover which variables that it specified when calling *BSAInit*( ) were interpreted by the Backup Service. Since the XBSA Application is in control, it can always abort a session if it disagrees with the Backup Service about the environment.

When an XBSA Application connects to a Backup Service, it can make an initial call to *BSAQueryApiVersion*( ) to determine the highest version of the specification supported. If the client supports that version, it should specify it when calling *BSAInit*( ). If the client 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 encounterered, and the BSA Application supports other versions, it may retry the call to *BSAInit*( ) specifying a different version.

All XBSA Application must set the BSA_API_VERSION environment variable in the environment structure used for the BSAInit() call. XBSA Applications that do not set this variable are assumed to be version 0.1.X clients by the Backup Service. The Backup Service will either reject such XBSA Application or provide a 0.1.X compatible API.

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. The **BSA_SecurityToken** or the **BSA_ObjectOwner** is invalid.

[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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

[BSA_RC_VERSION_NOT_SUPPORTED]
The Backup Service implementation does not support the specified version of the interface.

NAME

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 XBSA. The version information consists of the issue, version within the issue, and level within the version. If the Backup Services implementation supports more than one version, the latest version information will be 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 **apiVersion** pointer was encountered.

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSAQueryObject — query about object copies

SYNOPSIS

```
#include <xbsa.h>

int BSAQueryObject(long bsaHandle,
                   BSA_QueryDescriptor *queryDescriptorPtr,
                   BSA_ObjectDescriptor *objectDescriptorPtr)
```

DESCRIPTION

The *BSAQueryObject*( ) call initiates a request for information on object copies (for example, backup or archive) from a Backup Service. The results of the query will be determined by the search conditions specified in the queryDescriptor. The object descriptor for the first object satisfying the query search conditions is returned in the BSA_ObjectDescriptor (referenced by the BSA_ObjectDescriptorPtr parameter). The application can obtain the other object descriptors by successive calls to *BSAGetNextQueryObject*( ).

PARAMETERS

long *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 Object descriptor for the first 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 wild-card capability is available as follows:

Data Type	**Wild-card Options**
string	"*" matches 0 or more characters "?" matches exactly one character "\\*" matches a literal "*" "\\?" matches a literal "?" "\\\\" matches a literal "\\"
time	zero value = any time
enumerations	ANY value matches any value
BSA_ObjectOwner	defaults to value specified at session initialization

The following examples illustrate wild-card string matching:

BSA_ObjectName.pathName = /server/*	would match all objects for this server
BSA_ObjectName.pathName = /server/rootdbs/*	would match all levels of rootdbs
BSA_ObjectName.pathName = /server/????	would match all levels whose name is exactly 4 characters long

String matching is performed without any interpretation of the string contents. There is no implied knowledge of the structure of the string contents.

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 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 objects matched the given query.

[BSA_RC_NULL_ARGUMENT]
A NULL pointer was encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSAQueryServiceProvider — retrieve a string identifying the Backup Service provider

SYNOPSIS

```
#include <xbsa.h>

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

DESCRIPTION

The *BSAQueryServiceProvider*( ) call returns a hierarchical string identifying the Backup Service provider. The content of the string is implementation dependent.

PARAMETERS

BSA_UInt32 **sizePtr* (I / O)
This parameter contains the size of the *provider* buffer in bytes.

char **delimiter* (O)
This parameter points to the character that is used to delimit fields in the provider hierarchical string.

char **providerPtr* (O)
This parameter points to a data area that contains an implementation-dependent hierarchical string which conveys information identifying the Backup Service provider.

EXTENDED DESCRIPTION

The format of the *provider* string is the same as that of the BSA_SERVICE_PROVIDER environment variable (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, this will force a [BSA_RC_BUFFER_TOO_SMALL] error, thus providing 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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSASendData — send a byte stream of data in a buffer

SYNOPSIS

```
#include <xbsa.h>

int BSASendData(long bsaHandle, BSA_DataBlock32 *dataBlockPtr)
```

DESCRIPTION

BSASendData( ) sends a byte stream of data to a Backup Service in a buffer. The calling application can pass any data for storage in the XBSA system. *BSASendData*( ) can be called multiple times, in case the byte stream of data to be sent is large. This call may be used only after a *BSACreateObject*( ) or another *BSASendData*( ) call.

PARAMETERS

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

BSA_DataBlock32 **dataBlockPtr* (I)
This parameter points to a structure that includes both a pointer to the buffer from which the data is to be sent, as well as the size of the buffer.

EXTENDED DESCRIPTION

The Backup Service may not overwrite any of the fields in the **BSA_DataBlock32** structure. The Backup Service may write into the header and trailer portions of the buffer.

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 encounterered in one of the arguments

[BSA_RC_SUCCESS]
The function succeeded.

NAME

BSATerminate — terminate a session

SYNOPSIS

```
#include <xbsa.h>

int BSATerminate(long bsaHandle)
```

DESCRIPTION

The *BSATerminate*( ) call terminates the session with the Backup Service that was set up by a previous *BSAInit*( ) call and is associated with the bsaHandle. It also releases any resources acquired to set up the environment for the session. The Backup Service should allow a grace period before deallocating devices or rewinding tapes. If *BSATerminate*( ) is called within a transaction, the transaction will be aborted.

PARAMETERS

long *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.

Chapter 5

Type Definitions and Data Structures

This chapter describes the type definitions and data structures used in the XBSA interface. They are defined in a C Language Header File **<xbsa.h>**. A sample file is contained in Appendix B.

5.1 Type Definitions

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

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;`
		`/* or a single simple type if the target`
		` * platform supports a simple type that`
		` * yields a 64 bit quantity`
		` */`

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;`
		`/* or a single simple type if the target`
		` * platform supports a simple type that`
		` * yields a 64 bit quantity`
		` */`

Shared Memory Buffer reference	BSA_ShareId	<operating system dependent>

Table 5-1 Type Definitions

The 3rd column in the table above is not mandatory. The type definitions must be true to the what the type name specifies.

For 64 bit architectures, and architectures that support 64 bit quantities as a simple type, the BSA_UInt64 type may not be a structure.

5.2 Enumerated Types

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

5.2.1 BSA_CopyType

The BSA_CopyType enumeration describes the type of the operation used to create the 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:

Constant	Value	Definition
ANY	1	Used for matching any copy type (for example, ‘‘backup’’ or ‘‘archive’’ in the copy type field of structures for selecting query results).
ARCHIVE	2	Specifies that the copy type should be ‘‘archive’’.
BACKUP	3	Specifies that the copy type should be ‘‘backup’’.

Table 5-2 BSA_CopyType Enumeration Values

5.2.2 BSA_ObjectStatus

The BSA_ObjectStatus enumeration describes the current status of the 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:

Constant	Value	Definition
ANY	1	Provides a wild card 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.

Table 5-3 BSA_ObjectStatus Enumeration Values

5.2.3 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:

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’’.

Table 5-4 BSA_ObjectType Enumeration Values

5.2.4 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
} BSA_Vote;
```

The meaning of the enumeration values is as follows:

Constant	Value	Definition
COMMIT	1	The transaction is to be committed.
ABORT	2	The transaction is to be aborted.

Table 5-5 BSA_Vote Enumeration Values

5.3 Constant Values

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

Constant	Value	Definition
[BSA_ANY]	1	General-purpose enumeration wild-card 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 resource mgr name length
[BSA_MAX_TOKEN_SIZE]	64	Max size of a security token

Table 5-6 XBSA Constants and Values

5.4 Data Structures

5.4.1 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:

issue	Issue Number of the XBSA Specification
version	Version Number of the XBSA Specification
level	Implementation-defined version number

For implementations of the XBSA Preliminary Specification (Document Number P424), the values of the **BSA_ApiVersion** structure were implementation-dependent. For implementations of the XBSA Technical Standard (this document — C425), the values should be 1,1,X:

Published Document	Issue	Version	Level
Preliminary Specification (P424)	*	*	*
Technical Standard (C425)	1	1	X

* = implementation-dependent

Table 5-7 BSA_ApiVersion Structure Values

5.4.2 BSA_DataBlock32

The **BSA_DataBlock32** structure is used to pass data between an XBSA Application and the Backup Service. 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:

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

Table 5-8 BSA_DataBlock32 Structure Usage

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

```
bufferLen    >= headerBytes + numBytes
trailerBytes == (bufferLen - numBytes - headerBytes)
```

The layout of the buffer is as follows:

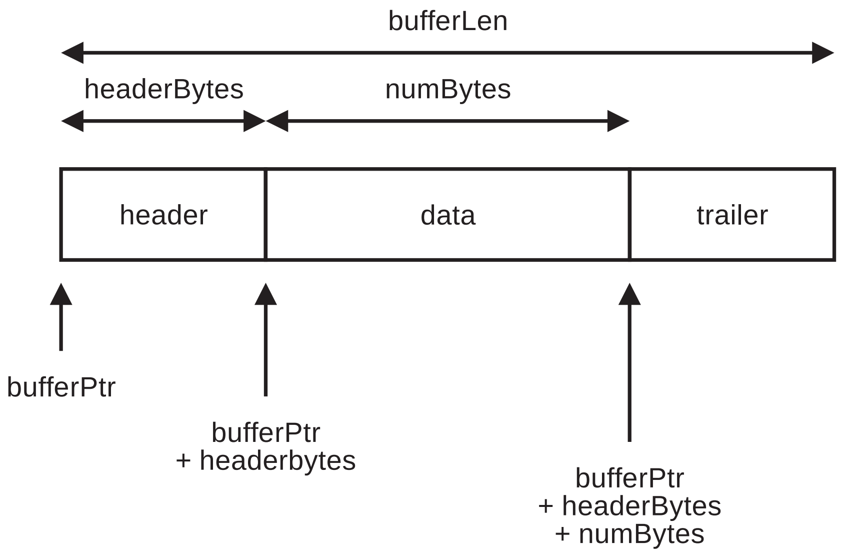

Figure 5-1 BSA_DataBlock32 Buffer Layout

The header and trailer portions of the buffer are reserved for the use of the Backup Service, 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 Backup Service are obtained by calling *BSACreateObject*( ) or *BSAGetObject*( ).

5.4.3 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[MAX_RC_OBJECTDESCRIPTION];
    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 Backup Service (Direction = out). Some fields are optional.

The usage of the structure fields is defined as follows:

Field Name	Definition	Supplied By	Status
rsv1	reserved field	-	-
objectOwner	Owner of the object	client	optional
objectName	Object name	client	mandatory
createTime	Create time	service	mandatory
copyType	Copy type: archive or backup	client	mandatory
copyId	Unique object identifier	service	mandatory
restoreOrder	Provides hints to the XBSA Application that allow it to optimize the order of object retrieval requests	service	optional
rsv2	reserved field	-	-
rsv3	reserved field	-	-
estimatedSize	Estimated object size in bytes, may be up to (2ˆ64 - 1) bits	client	mandatory
resourceType	for example, UNIX file system	client	mandatory
objectType	for example, file, directory, database	client	mandatory
objectStatus	Most recent / Not most recent	service	mandatory
rsv4	reserved field	-	-
objectDescription	Descriptive label for the object	client	optional
objectInfo	Application-specific information	client	optional

Table 5-9 BSA_ObjectDescriptor Structure Usage

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 all zeros (that is, a zero-filled field).

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.

5.4.4 BSA_ObjectName

The **BSA_ObjectName** structure is the name assigned by an XBSA Application to an 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:

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

Table 5-10 BSA_ObjectName Structure Usage

An **objectSpaceName** is an optionally defined, fixed-length character string. It identifies a logical space, called an Object space, in 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 concept of an Object space is used to provide a primary grouping of XBSA Objects, which may be used for object search by a user and/or for object management by the Backup Service. Additional groupings are provided by Filespec and by object attributes. Examples of an **objectSpaceName** are *C: Drive* and *VolumeLabel=XYZ*.

A **pathName** is a hierarchical character string that identifies an XBSA Object within an ObjectSpace.

An example of a **pathName** for the backup copy of a UNIX file may be its original path name and file name, for example, */documents/opengroup/backup.proposal*.

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

5.4.5 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:

Field Name	Definition
bsa_ObjectOwner	this is the name that the Backup Service authenticates
app_ObjectOwner	this is the name defined by the application

Table 5-11 BSA_ObjectOwner Structure Usage

The **bsa_ObjectOwner** identifies an XBSA Application (for example, an XBSA Application associated with a UNIX file system located at a certain node).

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

The **app_ObjectOwner** may 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 Backup Service. Its registration and authentication is the XBSA Application's responsibility. Examples of a typical **app_ObjectOwner** are *Smith*, *AccountingDept.Clerk1* and * (unspecified).

5.4.6 BSA_QueryDescriptor

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

```
#include <time.h>;
typedef struct {
    BSA_ObjectOwner     objectOwner;
    BSA_ObjectName      objectName;
    struct tm           rsv1;
    struct tm           rsv2;
    struct tm           rsv3;
    struct tm           rsv4;
    BSA_CopyType        copyType;
    char                rsv5[31];
    char                rsv6[31];
    char                rsv7[31];
    BSA_ObjectType      objectType;
    BSA_ObjectStatus    objectStatus;
    char                rsv8[100];
} BSA_QueryDescriptor;
```

The usage of the structure fields is defined as follows:

Field Name	Definition
objectOwner	Owner of the object
objectName	Object name
rsv1	reserved field
rsv2	reserved field
rsv3	reserved field
rsv4	reserved field
copyType	Copy type: archive or backup
rsv5	reserved field
rsv6	reserved field
rsv7	reserved field
objectType	for example, file, directory, database
objectStatus	Most recent / Not most recent
rsv8	reserved field

Table 5-12 BSA_QueryDescriptor Structure Usage

5.4.7 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];
```

Appendix A

Information for Backup Services Developers

This appendix is not a normative part of the specification.

A.1 Networked Environments

As an implementation example, a specific Backup Service may be provided for a standalone system as a local XBSA subsystem, which supports the Open Systems Backup Services API directly. Alternatively, as an example for a distributed environment, the Backup Service may be provided by one or more *BSA Server(s)* on a network, and a *BSA Client* implementation may reside on each Client system, supporting the Open Systems Backup Services API and communicating with the XBSA Server(s) using a suitable communication facility and protocol — see Figure A-1.

The architecture supports heterogeneous networks and any suitable communication protocol.

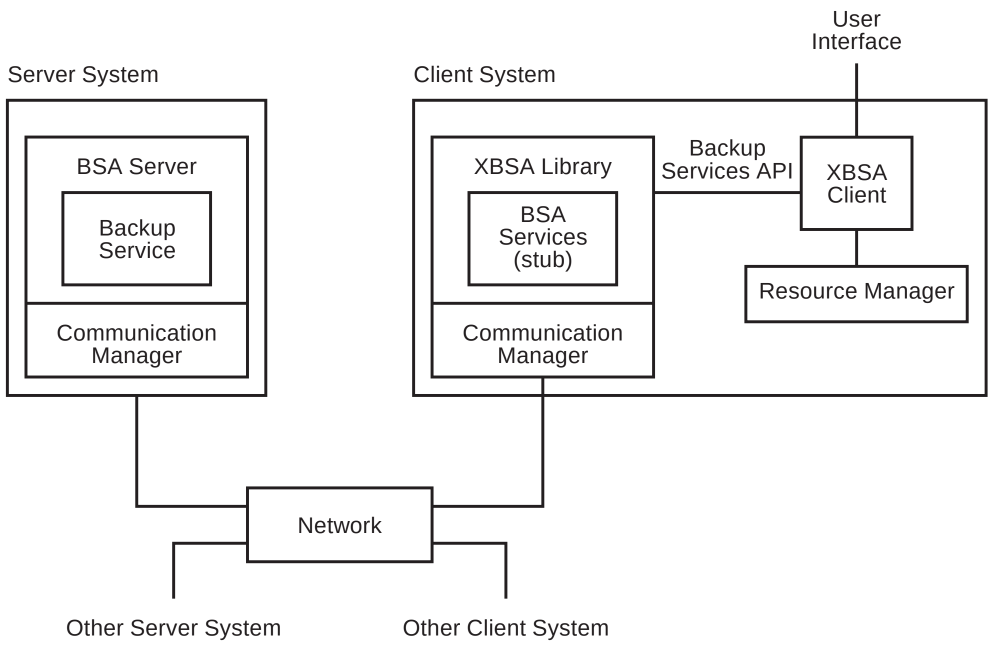

Figure A-1 An Example of a Distributed Backup System

A.2 Storage Hints

Although not defined in the specification, the **objectSpaceName** field in the **BSA_ObjectName** structure may be used to obtain hints on how to store the object's data.

A.3 Object Routing

Backup Service implementations may assume that process environment variables can be passed via the shell or command interpreter to the XBSA Client, and from there to the Backup Service to aid the Backup Service in selecting the appropriate backup medium for a session. There are a couple of hidden assumptions behind this - that the program called by the user is the XBSA Client that connects to the Backup Service and that the program is called in such a way that the environment variables can be set by the Backup Service itself. Another flaw in this approach is the presumption that the user is willing or knowledgeable enough to set these variables properly for each backup or restore event in the case that the Client is not invoked by the Backup Service. A third flaw is that all of this knowledge is non-transferable from one XBSA and Backup Service implementation to another. Together, these impose a significant experience or training requirement on the user.

The same problems hold if the user is expected to pass command line arguments to the XBSA Client. The Backup Service should not rely on these mechanisms to transfer routing information through the XBSA Client. Instead, the Backup Service should determine how to route XBSA Object Data by inspecting each XBSA Object individually. Several solutions are possible:

Round-Robin Solution

One possible, but inadequate, solution is for the Backup Service or XBSA implementation to simply route objects to the next-available device. This allows for parallel backups, but at the cost of poor sequencing of objects on backup media and of poor predictability.

Session-Guessing Solutions

A second solution is for the XBSA implementation to try to guess which set of objects belong together and send these to one set of devices, while another set of objects grouped together would be sent to a different set of devices. This takes care of the predictability problem and sequencing problem, but only if the guess is correct.

Some proposed session identification schemes have been based on either timing of backup events, the process ids of the XBSA Clients, or BSAInit/BSATerminate function call pairs. In all of these cases, the XBSA implementation has to make possibly false assumptions about the Client's behavior. For example, if a database administrator starts a database backup at the same time that a system administrator starts a filesystem backup, then a timing-based session guess will combine these completely separate events into a single session. BSAInit/Terminate pairs are also unreliable because a single user request may be distributed across several processes, each of which is required to call *BSAInit*( ) to get started.

Object Descriptor Solution

A third mechanism for routing objects is to actually use the metadata passed through XBSA to choose the set of devices or media to which an object should be written. The XBSA implementation or Backup Service can parse the BSA_ObjectDescriptor passed in the *BSACreateObject*( ) call. Whether it intelligently uses the contents or simply compares each field with a device mapping provided by the user, this general scheme provides great flexibility. For example, all objects created with a particular **resourceType** could be sent to the same medium. The Backup Service could allow the administrator to create a configuration file that lists groups of **resourceType** values and maps them to a particular group of devices or media; a newly created object would then cause the Backup Service to scan the configuration file for this resourceType to find the right backup device or medium to use.

This solution does not address grouping of objects into larger entities. The Backup Service has to make these decisions based on the call sequence and routing. It is up to the Backup Service to group objects, if possible, for performance considerations.

Disadvantages of this approach are that devices cannot be preallocated and media cannot be premounted as the decision of where to put an object is made in the *BSACreateObject*( ) call, and that ad-hoc backups may require special steps by the user to temporarily reconfigure the object routing.

Advantages are that the base configuration can be created once, with only occasional maintenance, and routing is extremely predictable. All fields in the BSA_ObjectDescriptor could be used. The routing of backup objects then becomes dependent on the configurability of the storage manager and the accuracy and completeness of the BSA_ObjectDescriptor created by the Backup Client. Either the Backup Service, XBSA implementation, or XBSA Client can then be modified or upgraded without affecting the others' ability to function.

A.4 Restore Order

The **restoreOrder** field in the **BSA_ObjectDescriptor** is assigned by the Backup Service at the time the XBSA Object is created. Restore order values provide a hint to the XBSA Client as to which sequence of *BSAGetObject*( ) calls would make most efficient use of the Backup Service's resources. The XBSA Client is free to ignore these values and the Backup Service has no obligation to completely optimize its assignment of **restoreOrder**. Multiple objects with the same **restoreOrder** indicate that the Backup Service has no preference for which of them gets processed first. If the XBSA implementation or the Backup Service do not know how to provide these hints, then the restoreOrder should be left with the value 0/0. Restore order is assigned at object creation time.

To make best use of the restore order suggested for a group of objects, the XBSA Client should call *BSAQueryObject*( ) and *BSAGetNextQueryObject*( ) for as many of the objects to be restored as possible. It should, within the requirements imposed by the application, restore the objects in ascending restoreOrder sequence. The **restoreOrder.left** is the primary sort key, and **restoreOrder.right** is the secondary sort key. If N objects are to be restored at the same time, then the first N objects in ascending sort order should be the ones chosen by the Client.

The values for **restoreOrder** can be determined in several ways, based on several criteria. One possible mechanism for selecting values is to use dates - assign the **restoreOrder.left** to the current year, and **restoreOrder.right** to the number of seconds since midnight, January 1, of the current year. A restore would then sort objects by the time at which they were backed up.

Another method might be to assign each device or medium a unique integer, which gets inserted into the **restoreOrder.right** for every object that is backed up on that device or medium, and put the position of that object on the device or medium in the **restoreOrder.left**. A restore could then read the first object on each medium first, then move on to the second object on each medium, and so on. If there are as many devices as there are media, then parallel operations would be quite efficient.

C Language Header File

```
/* xbsa.h
 *
 * This is a sample C header file describing the XBSA.
 *
 * This appendix is not a normative part of the
 * specification and is provided for illustrative
 * purposes only.
 *
 * Implementations must ensure that the sizes of integer
 * datatypes match their names, not necessarily the typedefs
 * presented in this example.
 *
 */

#ifndef _XBSA_
#define _XBSA_

#include <time.h>

/* BSA_Int16
 */
typedef short BSA_Int16;

/* BSA_Int32
 */
typedef int BSA_Int32;

/* BSA_Int64
 */
typedef struct { /* defined as two 32-bit integers */
    BSA_Int32   left;
    BSA_Int32   right;
} BSA_Int64;

/* BSA_UInt16
 */
typedef unsigned short BSA_UInt16;

/* BSA_UInt32
 */
typedef unsigned int BSA_UInt32;

/* BSA_UInt64
 */
typedef struct { /* defined as two unsigned 32-bit integers*/
    BSA_UInt32  left;
    BSA_UInt32  right;
} BSA_UInt64;

/* BSA_ShareId
 */
typedef BSA_ShareId /* operating system dependent*/
```

```
/* Constants used
 *
 * Maximum string lengths (lower bound), including trailing null
 */
#define BSA_MAX_APPOBJECT_OWNER      64
#define BSA_MAX_BSAOBJECT_OWNER      64
#define BSA_MAX_DESCRIPTION         100
#define BSA_MAX_OBJECTSPACENAME    1024
#define BSA_MAX_OBJECTINFO          256
#define BSA_MAX_PATHNAME           1024
#define BSA_MAX_RESOURCETYPE         31
#define BSA_MAX_TOKEN_SIZE           64

/* Other constants */

#define BSA_ANY                      1

/*
 * Return Codes Used
 */
#define BSA_RC_ABORT_SYSTEM_ERROR          0x03
#define BSA_RC_ACCESS_FAILURE              0x4D
#define BSA_RC_AUTHENTICATION_FAILURE      0x04
#define BSA_RC_BUFFER_TOO_SMALL            0x4E
#define BSA_RC_INVALID_CALL_SEQUENCE       0x05
#define BSA_RC_INVALID_COPYID              0x4F
#define BSA_RC_INVALID_DATABLOCK           0x34
#define BSA_RC_INVALID_ENV                 0x50
#define BSA_RC_INVALID_HANDLE              0x06
#define BSA_RC_INVALID_OBJECTDESCRIPTOR    0x51
#define BSA_RC_INVALID_QUERYDESCRIPTOR     0x53
#define BSA_RC_INVALID_VOTE                0x0B
#define BSA_RC_NO_MATCH                    0x11
#define BSA_RC_NO_MORE_DATA                0x12
#define BSA_RC_NULL_ARGUMENT               0x55
#define BSA_RC_OBJECT_NOT_FOUND            0x1A
#define BSA_RC_SUCCESS                     0x00
#define BSA_RC_TRANSACTION_ABORTED         0x20
#define BSA_RC_VERSION_NOT_SUPPORTED       0x4B

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

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

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

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

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

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

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

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

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;

typedef struct {
    BSA_ObjectOwner     objectOwner;
    BSA_ObjectName      objectName;
    struct tm           rsv1;
    struct tm           rsv2;
    struct tm           rsv3;
    struct tm           rsv4;
    BSA_CopyType        copyType;
    char                rsv5[31];
    char                rsv6[31];
    char                rsv7[31];
    BSA_ObjectType      objectType;
    BSA_ObjectStatus    objectStatus;
    char                rsv8[100];
} BSA_QueryDescriptor;

typedef BSA_SecurityToken   char[BSA_MAX_TOKEN_SIZE];

/* Function Prototypes
 */

int BSABeginTxn (long);

int BSACreateObject(long, BSA_ObjectDescriptor *, BSA_DataBlock32 *);

int BSADeleteObject(long, BSA_UInt64);

int BSAEndData(long);

int BSAEndTxn(long, BSA_Vote);

int BSAGetData(long, BSA_DataBlock32 *);

int BSAGetEnvironment(long, BSA_ObjectOwner *, char **);

int BSAGetLastError(BSA_UInt32 *, int *);

int BSAGetNextQueryObject(long, BSA_ObjectDescriptor *);

int BSAGetObject(long, BSA_ObjectDescriptor *, BSA_DataBlock32 *);

int BSAInit(long *, BSA_SecurityToken *, BSA_ObjectOwner *, char **);

int BSAQueryApiVersion(BSA_ApiVersion *);

int BSAQueryObject(long, BSA_QueryDescriptor *, BSA_ObjectDescriptor *);

int BSAQueryServiceProvider(BSA_UInt32 *, char *, char *);

int BSASendData(long, BSA_DataBlock32 *);

int BSATerminate(long);

#endif
```

Glossary

active object
An object that is accessed frequently (as defined by an administration policy) by the application or service.

application object
The objects managed by an application or service (for example, file systems, documents) which will be backed-up or archived using the Open Backup Services API.

app_ObjectOwner
An optionally-supplied name, which is combined with an **bsaObjectOwner** to make the **ObjectOwner** which identifies a user of XBSA to a Backup Service. Used for access control.

archive
A copy of an Application Object generally made for long-term, low-usage storage purposes. Typically the Application Object is deleted from the application's repository after the copy is made.

backup
A copy of an Application Object generally made for the purpose of recovery in the event of system failure or human error. Typically the Application Object is retained in the application's repository after the copy is made.

backup service
An implementation of the lower level of XBSA which responds to XBSA Managers or XBSA Clients requesting services.

bsa_ObjectOwner
The name of the application or service which is using XBSA, and which has been registered with a particular Backup Service.

Catalog
Storage used by the Backup Service to hold object cataloging information (*metadata*).

Copy Id
A unique integer identifying a particular instance of a stored object.

Copy Type
An XBSA Object attribute with values of BSACopyType_BACKUP or BSACopyType_ARCHIVE.

handle
An identifier issued to an application or service by the Backup Service when a particular session is initiated and authenticated.

hierarchical character string
A variable-length character string containing delimited fields which are successively examined using a search pattern. Used to establish access rules, searches, and so on.

inactive object
An object that is accessed infrequently (as defined by an administration policy) by the application or service.

incremental backup
An application-specific behavior causing selective backup of objects to occur according to an applied rule.

object descriptor
A collection of object attributes, containing metadata of an XBSA object. Stored in an XBSA Catalog and associated with an individual Blob.

object type
An XBSA Object attribute, assigned by the using application or service. An enumerated integer value (see Section 5.2.3 on page 63).

ObjectName
The name assigned by the XBSA User to an XBSA Object that is unique within the XBSA User's domain, consisting of two parts: **objectSpaceName** and **pathName**.

ObjectOwner
The name of the owner of an XBSA Object, consisting of two parts: **bsa_ObjectOwner** and **app_ObjectOwner**.

Objectspace
The name of a logical space in the user's domain where an object resides.

objectSpaceName
A variable-length string that identifies an Objectspace; examples are: *C:Drive*, *VolumeLabel=XYZ*.

pathName
A hierarchical string that further identifies an XBSA object within an Objectspace. Example: */documents/xopen/backup.proposal*.

restore
The operation of obtaining a copy of a currently active object from the Backup Service, and placing this copy back into the application's domain to correct a system or human failure.

Retention Period
An indication of how long a backup object is to be kept.

retrieve
The operation of obtaining a copy of a currently inactive object from the Backup Service, and placing this copy back into the application's domain so that the data may be accessed.

Status
An XBSA Object attribute with values of BSAObjectStatus_MOST_RECENT or BSAObjectStatus_NOT_MOST_RECENT.

XBSA
The Open Backup Services API (this document).

XBSA application
An XBSA Client or XBSA Manager.

XBSA client
Application-specific software which uses XBSA to request services from the Backup Service on behalf of a particular application. Typically this XBSA Client is tightly bound to a user application (such as a DBMS) or an operating system service (such as a file system) by existing in the address space of the application/service or being packaged with this function.

XBSA manager
Management software which uses XBSA to manage the services provided in Backup Services. Typically this XBSA Manager may manage the operation of a variety of Backup Service implementations from a variety of vendors.

XBSA object
An object as viewed through the XBSA interfaces. Has a two-part name: **ObjectOwner** and **ObjectName**.

Repository
Storage used by the Backup Services to hold the (uninterpreted) data making up an object which is managed.

XBSA session
A logical connection between an XBSA Application and a Backup Service, delimited by calls to *BSAInit*( ) and *BSATerminate*( ).

Index