Quantum

Storage Manager API Guide

StorNext API 2.0.3

6-01375-07 Rev A

Quantum StorNext API Guide, 6-01375-07 Rev A, July 2012, Product of USA.

Contents

Chapter 1

Introduction

About This Guide

This guide contains information and instructions necessary to install and use StorNext API (SNAPI). This guide is intended for system administrators, programmers, and anyone interested in learning about installing and using the StorNext APIs.

The SNAPI Storage Manager API Guide is divided into the following chapters:

- StorNext APIs
- API Example

StorNext API contains StorNext Storage Manager APIs which can be used to make calls to third-party applications, resulting in enhanced operations between third-party applications and StorNext.

Note: StorNext File System APIs are documented in the StorNext File System API Guide.

Product Compatibility

Refer to the compatibility matrix at the following location for the latest SNAPI compatibility information:

http://www.quantum.com/ServiceandSupport/SoftwareandDocumentationDownloads/SNMS/Index.aspx

Installing StorNext APIs

The StorNext Storage Manager APIs on the CD must be installed before you can call them. For installation instructions, see Installing the APIs on the Storage Manager Host in the chapter StorNext APIs.

Running APIs Remotely

Beginning with SNAPI 2.0, you can now run StorNext Storage Manager APIs remotely from a client. Functionally, operation from a remote client is identical to running the APIs locally. If you want to use this feature, remote clients must be installed and configured as described in Installing the APIs on a Remote Client and Configuring SNAPI in the chapter StorNext APIs.

Explanation of Warnings, Cautions and Notes

The following cautions, and notes appear throughout this document to highlight important information.

Caution: Indicates a situation that may cause possible damage to equipment, loss of data, or interference with other equipment.

Note: Indicates important information that helps you make better use of your system.

Quantum Service and Support

More information about this product is available on the Quantum Service and Support website at www.quantum.com/ServiceandSupport. The Quantum Service and Support website contains a collection of information, including answers to frequently asked questions (FAQs). You can also access software, firmware, and drivers through this site.

For further assistance, or if training is desired, contact the Quantum Technical Assistance Center:

North America	+1-800-284-5101 (toll free) +1-720-249-5700
EMEA	+00-800-7826-8888 (toll free) +49-6131-3241-1164
APAC	+1-800-7826-8887 (toll free) +603-7953-3010
Online Service and Support	www.quantum.com/OSR
Worldwide Web	www.quantum.com/ServiceandSupport

(Local numbers for specific countries are listed on the Quantum Service and Support Website.)

Chapter 2
StorNext APIs

Introduction

This chapter describes the application programming interfaces (APIs) available for StorNext Storage Manager.

This chapter contains the following major topics:

Note: You must install the Storage Manager APIs from the CD by following the installation instructions in Installing the APIs on the Storage Manager Host on page 6.

Throughout this chapter, the terms library and archive are used. Library refers to an actual physical library (e.g., with robotics), whereas archive is a more general term that may refer to a library, a vault, or anywhere else where media can be stored.

A vault is a logical archive type. It is any location that contains copies of media that have been removed from the operational system. Since a vault has no robotics, if a vault has tape drives, tape media must be manually mounted to the drive and dismounted from the drive.

Installing the APIs on the Storage Manager Host

This section describes how to install the SNAPI Storage Manager APIs and the SNAPI Server on the StorNext Storage Manager host. After installation is complete, the SNAPI Server will listen for requests from all local and remote SNAPI clients.

The SNAPI Server will log operational and error messages to /usr/adic/SNAPI/logs/. The SNAPI Server will also handle all local and remote requests from snclix, the SNAPI command line interface utility.

Installation instructions differ slightly depending on the delivery mechanism used:

- Installation from CD (not applicable to SNAPI 2.0.3)
- Installation from a Web download
- Installation from a Download Media Kit

Be sure to read the text associated with the SNAPI delivery mechanism you plan to use.

After a point the installation process is the same regardless of delivery mechanism. Delivery-method specific differences are noted in the procedure.

About Installing SNAPI from CD

Previous SNAPI releases were installed from CD, and future SNAPI releases may support this delivery mechanism as well.

However, SNAPI 2.0.3 is available only as a Web release or Download Media Kit.

About Installing SNAPI from a Web Download

First, you will download an executable program that will require you to accept the Quantum StorNext End User License Agreement (EULA), and then you will extract the SNAPI installer.

As part of the installation from a Web download, the software files are extracted by default into the `/tmp/stornext` directory. The software, when extracted, requires approximately 40 MB of space. Make sure there is enough free space in the default directory to extract the files.

If there is not sufficient space, you may need to specify an alternative directory with the -d option. To specify an alternate location before beginning the installation, use the following command: `# ./ <installation file> ?d <dir>`

where `<installation file>` is the name of the installation file, and`<dir>` is the directory into which you want to extract the installation files.

At this point you are ready to resume installation as described in Installing SNAPI.

About Downloading a SNAPI Media Kit (Creating an ISO Image)

Another way to obtain a StorNext API media kit is by downloading the software. This process consists of obtaining a download authorization certificate, creating a download account (or signing on if you already have an account), downloading a software ISO image onto DVD, and then installing the software as described in this guide.

For more information about downloading StorNext API media kits or to obtain a download authorization certificate, contact your Quantum StorNext support representative.

Installing SNAPI

Follow these steps to install SNAPI.

1. Log on as root.
2. (For CD installations only:) Insert and mount the product CD that contains the StorNext Storage Manager APIs.
3. Use the cd command to navigate to the directory that corresponds to your operating system:
   - RedHat50AS_26x86_64
   - RedHat60AS_26x86_64

- SuSE100ES_26x86_64
- SuSE110ES_26x86_64

4 From the operating system directory, run the program **`install.snapi`**.

Note: The **`install.snapi`** command must be run on the primary node. Assume that NodeA is initially the primary and NodeB is the secondary. Run **`install.snapi`** on the primary system (NodeA). Then stop cvfs on the primary to cause a failover to the secondary system (NodeB). When NodeB becomes primary, complete the snapi install/upgrade by running **`install.snapi`** on NodeB.

---

5 This program automatically decompresses the APIs and copies them to your StorNext directory in the following location: /usr/adic/SNAPI.

Note: Before installation begins, the StorNext API End User License Agreement (EULA) is displayed. You must accept the EULA before you can proceed with the installation.

---

6 After installing the Storage Manger APIs, in order to successfully compile and link with your applications you need the location of the main header file and the libraries Quantum delivers.

The main header file location: `/usr/adic/SNAPI/inc/API.hh`

The dynamic library location: `/usr/adic/SNAPI/lib/libsnapi.so`

The static library location: `/usr/adic/SNAPI/lib/static/libsnapi.a`

Installing the APIs on a Remote Client

This section describes how to install the SNAPI Storage Manager APIs on a remote client connected to a SNAPI Server host. As a prerequisite, SNAPI must have already been installed on the Storage Manager host as described in Installing the APIs on the Storage Manager Host.

To install a remote client:

1 Copy the script /usr/adic/SNAPI/bin/install.snapiclient from the Storage Manager host to the remote host.

Note: To ensure that the install.snapiclient script runs properly, before proceeding verify that RCP is enabled between the remote client host and the StorNext Storage Manager host.

2 Run the install.snapiclient script on the remote host, passing it the name of the Storage Manager host. For example:
install.snapiclient <SNAPI_Server_Host>

Note: During the .bin file extraction, the StorNext API End User License Agreement (EULA) is displayed. You must accept the EULA before you can proceed with the installation.

The installation script also installs the snclix utility, which can be used to run SNAPI API commands on the command line.

Note: A SNAPI upgrade on a remote client is accomplished by running the installation procedure described above. To remove SNAPI from a remote client, simply remove all contents of the /usr/adic/SNAPI/ directory.

Configuring SNAPI

SNAPI configuration for both local and remote clients is accomplished by editing the /usr/adic/SNAPI/config/snapi.cfg file on the SNAPI server and remote hosts, respectively. The snapi.cfg file allows you to specify the following:

- SNAPI server names
- INET socket port value for remote clients
- Client timeout value. This value specifies how long the client should wait for a response from the SNAPI server. This value can be between 0-1000 minutes. The default value is 30 minutes.

You are not required to modify the snapi.cfg file before running the APIs unless you are enabling HA support for remote clients, or in other special cases. (See HA Failover Support for more information about enabling HA support.)

Snapi.cfg Example

The following example describes and illustrates each of the parameters in the snapi.cfg file. (You can find the example in the snapi.cfg.default file installed at /usr/adic/SNAPI/config/snapi.cfg.default.)

```
<!--   The serverName indicates the host on which the
       StorNext server is running. If this is an HA system,
       add a line to indicate the failover host. -->

<PARAMETER name="serverName" value="localhost"/>

<!--   The serverPort is used for remote client connections.
       There is no need to edit this value unless the port
       is already in use, in which case the port value on
       the StorNext server must be edited also. -->

<PARAMETER name="serverPort" value="61776"/>


<!--   The clientTimeOut is the maximum time (in secs) a
       client will spend attempting to fulfill a request to
       the server. Set the value to 0 to indicate the client
       should try indefinitely to fulfill the request. -->

<PARAMETER name="clientTimeOut" value="1800"/>
```

HA Failover Support

HA failover support applies only to remote clients, and is supported by the snapi.cfg file on the remote hosts. To enable HA support, specify the primary and secondary SNAPI server host names in an ordered list on separate lines in the /usr/adic/SNAPI/config/snapi.cfg file.

For example:

```
<PARAMETER name="serverName" value="zeus"/>

<PARAMETER name="serverName" value="hera"/>
```

Upon detecting failover, SNAPI internally will reissue the original request to the SNAPI server on behalf of the remote client. Connection attempts to the primary and secondary servers will continue until the client timeout limit is reached.

SNAPI Logs and Health Checks

Logs are available for the SNAPI server host. These logs use the existing StorNext logging framework, and are located at /usr/adic/SNAPI/logs/tac.

You can configure SNAPI logs by adjusting the parameters located at /usr/adic/SNAPI/logs/log_params.

Health checks are also available on the SNAPI server host. Health checks are also integrated with the existing StorNext Health Check framework.

To run health checks, from the StorNext home page, choose Health Check from the Service Menu. The Health Check Test screen appears. From this screen run one or both of these health checks:

- Network: This health check validates connections to the StorNext server
- Config: This health check validates the contents of the SNAPI configuration file (snapi.cfg)

Alternatively, you can accomplish the same thing manually by running the following tests:

- snapiverifyConnectivity: Verifies connection to the StorNext server. Located at /usr/adic/SNAPI/bin/snapiVerifyConnectivity.
- snapiverifyConfig: Validates the SNAPI configuration file (snapi.cfg) contents. Located at /usr/adic/SNAPI/bin/snapiVerifyConfig.

Use Cases

There are three different programmatic entry points for accessing the StorNext Storage Manager APIs:

- Using a C or C++ Library
- Using a C or C++ Library with XML Wrapper
- Using the SNAPI XML Command Line Interface (snclix)

This section contains examples showing how to use the APIs with these methods.

Note: Data types used in these APIs are defined in the header files located here: /usr/adic/SNAPI/inc/

Using a C or C++ Library

For each API, there is a corresponding C++ class with the same name as the API. Quantum provides both dynamic and static libraries, as well as a header file to link to the application program.

The following example illustrates typical API usage for this case.

```
#include <API.hh>

#include <sys/types.h>
#include <unistd.h>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;
using namespace Quantum::SNAPI;

int
main()
{
   Status status;

   try
   {
      // Create object for requesting file attributes.
      GetFileAttribute getFileAttrReq("/snfs/testFile.dat");

     // Send request to the server and get overall request status
     code.
      // Note: This method may throw exceptions (see catch block
     below).
      status = getFileAttrReq.process();

      // Check the returned status code.
      if (status.getCode() != SUCCESS)
      {
```

```
            cout << "StatusCode: " << status.getCodeAsString() <<
            endl;
            cout << "Description: " << status.getDescription() <<
            endl;
            cout << "LocalStatus.StatusCode: " <<
                   getFileAttrReq.getLocalStatus().getCodeAsString()
            << endl;
            cout << "LocalStatus.Description: " <<
                    getFileAttrReq.getLocalStatus().getDescription()
            << endl;
         }
         else
         {
            // Get the requested data.
            FileInfo fileInfo = getFileAttrReq.getFileInfo();

            cout << "FileName:        " << fileInfo.getFileName() <<
            endl;
            cout << "Location:        " <<
            fileInfo.getLocationAsString() << endl;
            cout << "Existing Copies: " <<
            fileInfo.getNumberOfExistingCopies() << endl;
            cout << "Target Copies:   " <<
            fileInfo.getNumberOfTargetCopies() << endl;

            MediaList media = fileInfo.getMedia();

            for (int i=0; i<media.size(); i++)
            {
               cout << "Media ID:        " << media[i] << endl;
            }
         }
      }
      catch (SnException& exception)
      {
         switch (exception.getCode())
         {
            case SUBFAILURE:
            case FAILURE:
            case SYNTAXERROR:
            default:
            {
               cout << "Exception code:   " << exception.what() <<
               endl;
```

```
            cout << "Exception detail: " << exception.getDetail()
            << endl;
         }
         break;
      }
   }
   catch (...)
   {
      cout << "Caught unknown exception." << endl;
   }

   return status.getCode();
}
```

Using a C or C++ Library with XML Wrapper

For this approach there is a single function called doXML available to the third-party program. This function's input and output are string type, and their contents are in XML format. Different APIs have different specifications on the XML input and output, as described for each API.

Note: It is your responsibility to provide an input string that satisfies the specification. Otherwise, a failure from the doXML function call might result.

The following example illustrates typical API usage for this case.

```
#include <API.hh>

#include <sys/types.h>
#include <unistd.h>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;
using namespace Quantum::SNAPI;

int
main()
{
   Status status;

   // Initialize input command in XML format.
```

```
   XML xmlIn("<?xml version=\"1.0\"?>"
             "<COMMAND name=\"GetFileAttribute\">"
             "   <ARGUMENT name=\"fileName\" value=\"/snfs/
             testFile.dat\"/>"
             "</COMMAND>");

   XML xmlOut;

   try
   {
      // Perform the request.
      status = doXML(xmlIn, xmlOut);
   }
   catch (SnException& excptn)
   {
      cerr << "Exception code:   " << excptn.what() << endl;
      cerr << "Exception detail: " << excptn.getDetail() << endl;
      status = excptn.getCode();
   }

   // Stream the results to standard out.
   cout << xmlOut << endl;

   return status.getCode();
}
```

Using the SNAPI XML Command Line Interface (snclix)

The snclix utility is a command line utility that provides an interface for SNAPI APIs from the SNAPI server and remote client hosts. This utility operates on XML input and output, and is very similar to the doXML() function. Both of these take an XML request definition as input, and return an XML response with requested information embedded in the response.

Note: It is your responsibility to generate a properly formatted XML request and to subsequently access the relative portions of information in the XML response.

Any XML information that is not relevant to the specified request will cause an error. This also applies to cases where single arguments are expected but multiple arguments are supplied.

XML character data must comply with Section 2.4 of the XML specification. In particular, when used in value strings, the following characters must be escaped using their ASCII numeric character references:

& = "&"

< = "<"

> = ">"

In addition, any non-printing characters (such as new line, tab, etc.) must be escaped using their ASCII numeric character references.

When using the snclix utility, after you create a file containing properly formatted XML (or optionally an input stream,) you then pass the information to the snclix executable program.

Internally, snclix calls the doXML() function to process the request and send the resulting XML response to an output file (or optionally streams it to stdout).

The snclix utility is located at /usr/adic/SNAPI/bin/snclix and has the following usage format:

```
snclix [-i infile] [-o outfile] [-c config]
```

`-i infile`: Specifies an input file. (File contents should be in XML format.)

`-o outfile`: Specifies an output file. (File contents will be in XML format.) If an outfile exits, it will be overwritten.

`-c config`: Specifies a configuration file.

Example of snclix Usage

```
snclix -i /tmp/GetFileAttributeRequest.xml -o /tmp/
GetFileAttributeResponse.xml
```

Where the /tmp/GetFileAttributeRequest.xml file contains the following input:

```
<?xml version="1.0"?>
<COMMAND name="GetFileAttribute">
<ARGUMENT name="fileName" value="/snfs/testFile.dat"/>
</COMMAND>
```

And the /tmp/GetFileAttributeResponse.xml file contains the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetFileAttribute" statusCode="0"
status="SUCCESS">
<FILEINFO statusCode="0" status="SUCCESS"
statusDescription="Command Successful">
    <INFO name="fileName" value="/snfs1/testFile.dat" />
    <INFO name="location" value="DISK AND TAPE" />
    <INFO name="numberExistingCopies" value="1" />
    <INFO name="numberTargetCopies" value="1" />
    <INFO name="mediaID" value="000082(1)" />
  </FILEINFO>
</RESPONSE>
```

API Descriptions and Arguments

A brief description is provided for each API, as well as its input arguments and output variables. Also provided for each API is an example showing XML input and output, and the corresponding C++ class declaration.

The following StorNext Storage Manager APIs are described in this section:

- Backup
- CheckoutMedia
- CleanMedia
- CopyMedia
- EjectMedia
- EnterMedia
- FileRetrieve
- GetArchiveCapacity
- GetArchiveList

- GetBackupStatus
- GetDriveList
- GetFileAttribute
- GetFileTapeLocation
- GetMediaList
- GetMediaStatus
- GetPolicy
- GetPortList
- GetSchedule
- GetSystemStatus
- MoveMedia
- PassThru
- RmDiskCopy
- SetArchiveState
- SetDirAttributes
- SetDriveState
- SetFileAttributes
- SetMediaState
- SetPolicy
- SetSchedule

For the C++ class declaration section, all API classes inherit from class Request, which has a public member function process(). The third-party program must call this function to have the API request processed. See the example code in Appendix B.

Note: The process() function should be called only once for each instantiation of an API object. API users should instantiate a new API object every time they need to call the process() function.

When you enter arguments for the StorNext Storage Manager APIs, wildcard characters such as the asterisk (*) are not supported. For example, when you run the FileRetrieve API you cannot enter '*' when entering the <filename> parameter.

Backup

This API initiates a backup operation.

Input

NA. This API has no command arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Initiate a system backup operation. -->
<COMMAND name="Backup"/>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="Backup" statusCode="0" status="SUCCESS"
              statusDescription="Backup initiation successful">
  <STATUSDETAIL statusCode="0" status="SUCCESS"
              statusDescription="Backup initiation successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class Backup : public Request
{
public:
   /// Default and Primary Contructor
   Backup();

   /// Method to return the local status
   const Status& getLocalStatus() const;
}
```

CheckoutMedia

This API allows you to check out media with the specified media IDs.

Input

mediaID: The ID of the media you want to check out. You can specify multiple media IDs to check out multiple media.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Checks out all specified media from the system. -->
<COMMAND name="CheckOutMedia">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- mediaID : valid media ID -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- One or more mediaID must be specified. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="mediaID" value="025311"/>
  <ARGUMENT name="mediaID" value="025312"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="CheckOutMedia" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
<!-- A status detail element will be returned for each media -->
 <STATUSDETAIL name="mediaID" value="025311"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
 <STATUSDETAIL name="mediaID" value="025312"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class CheckOutMedia : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// given a single media ID.
   CheckOutMedia(const std::string& inMediaID);

   /// Secondary Constructor
   /// This constructor is for instantiation of the object from a
  list of
   /// media ID to eject.
   CheckOutMedia(const MediaList& inMediaList);

   /// Method to return the list of media
   const MediaList& getMediaList() const;

   /// Method to return the collection of status pairs
   const StatusPairList& getLocalStatus() const;

   /// Method to return the status pair for a specific mediaID
   const StatusPair& getLocalStatus(const std::string& mediaID)
   const;
}
```

CleanMedia

This API cleans media by removing inactive files from the specified media.

Input

mediaID: The ID of the media you want to clean. You can specify multiple media IDs to clean multiple media.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Remove inactive files from all specified media. -->
<COMMAND name="CleanMedia">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- mediaID : valid media ID -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- One or more mediaID must be specified. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<ARGUMENT name="mediaID" value="025311"/>
<ARGUMENT name="mediaID" value="025312"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="CleanMedia" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <!-- A status detail element will be returned for each media -->
  <STATUSDETAIL name="mediaID" value="025311"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
  <STATUSDETAIL name="mediaID" value="025312"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class CleanMedia : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object.
   CleanMedia(const std::string& inMediaID);

   /// Secondary Constructor
   /// This constructor is intended for instantiation of the
   object from a
   /// vector of media ids.
   CleanMedia(const MediaList& inMedia);

   /// Method to return the list of media
   MediaList getMediaList() const;

   /// Method to return the list of local status pairs
   const StatusPairList& getLocalStatus() const;

   /// Method to return the specific local status pair for a given
   mediaID
   const StatusPair& getLocalStatus(const std::string& mediaID)
   const;
}
```

CopyMedia

This API copies the content of all specified media to a piece of blank media.

Input

mediaID: The ID of the media you want to copy. You can specify multiple media IDs to copy multiple media.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Copy the content of all specified media to blank media. -->
<COMMAND name="CopyMedia">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- mediaID : valid media ID -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- One or more mediaID must be specified. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<ARGUMENT name="mediaID" value="025311"/>
<ARGUMENT name="mediaID" value="025312"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="CopyMedia" statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
  <!-- A status detail element will be returned for each media -->
    <STATUSDETAIL name="mediaID" value="025311"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
    <STATUSDETAIL name="mediaID" value="025312"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class CopyMedia : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object.
   CopyMedia(const std::string& inMediaID);

   /// Secondary Constructor
   /// This constructor is intended for instantiation of the
   object from a
   /// vector of media ids.
```

```
    CopyMedia(const MediaList& inMedia);

    /// Method to return the list of media
    MediaList getMediaList() const;

    /// Method to return the list of local status pairs
    const StatusPairList& getLocalStatus() const;

    /// Method to return the specific local status pair for a given
   mediaID
    const StatusPair& getLocalStatus(const std::string& mediaID)
    const;
}
```

EjectMedia

This API moves the specified media from the archive to a media I/E slot. EjectMedia does an automated media eject. This API must be used with the MoveMedia API.

After running this API, you must call the EnterMedia API to physically move the media from the mailbox into the destination library, or in the case of a vault, to logically enter the media into the vault.

Input

portID: The port used for ejecting media. The port ID is in the format 0,0,15,###. Only the digits following the final comma are needed, indicated as ### in the format example. It is not needed for vault archives.

mediaID: The ID of the media to eject.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Eject a specific media from the system. -->
<COMMAND name="EjectMedia">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- portId : IE port id to eject media to -->
  <!-- mediaId : media to eject -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Exactly one each of the above arguments are required. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="portId" value="16"/>
  <ARGUMENT name="mediaId" value="012345"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="EjectMedia" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <STATUSDETAIL name="mediaID" value="012345"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class EjectMedia : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// given a port Id and a single media Id.
   EjectMedia(const std::string& inMediaId,
              const std::string& inPortId = "0");

   /// Method to set port Id
   void setPortId(const std::string& inPortId);

   /// Method to return the media Id
   const std::string& getMediaId() const;
```

```
   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}
```

EnterMedia

This API adds (inserts) media into the specified archive from the I/E slot. This API must be used after you run the EjectMedia API, and must be used with the MoveMedia API.

Input

archiveID: The archive ID to which media is added.

portID: The port used for entering media. The port ID is in the format 0,0,15,###. Only the digits following the final comma are needed, indicated as *###* in the format example. It is not needed for vault archives.

mediaID: The media ID to insert into the archive.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Enter a specific media into the system. -->
<COMMAND name="EnterMedia">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- archiveId : archive name to add media to -->
  <!-- portId : IE port id to move media from -->
  <!-- mediaId : media to add to archive -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Exactly one each of the above arguments are required. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="archiveId" value="archive2"/>
  <ARGUMENT name="portId" value="16"/>
```

```
   <ARGUMENT name="mediaId" value="012345"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="EnterMedia" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <STATUSDETAIL name="mediaID" value="012345"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class EnterMedia : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// given a archive Id, a port Id, and a single media Id.
   EnterMedia(const std::string& inArchiveId,
              const std::string& inMediaId,
              const std::string& inPortId = "0");

   /// Method to set portId
   void setPortId(const std::string& inPortId);

   /// Method to return the media Id
   const std::string& getMediaId() const;

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}
```

FileRetrieve

This API retrieves the specified single file from secondary storage to primary storage.

Input

filename: The pathname of the file to be retrieved.

newFileName (optional): The name you want to give the retrieved file. The default value is an empty string, which means the file data is retrieved to disk using the original file name. This argument is optional if used by itself, but required if you use startByte and endByte.

startByte (optional): The starting byte for a partial retrieval. The default value is 0, which means a full retrieval is expected. Do not use this argument (in conjunction with the endByte argument) if you use the *copyId* argument.

endByte (optional): The end byte for a partial retrieval. The default value is 0, which means a full retrieval is expected. Do not use this argument (in conjunction with the startByte argument) if you use the copyId argument.

modAccessTime (optional): TRUE or FALSE. Indicates whether the retrieve will modify the file's access time. The default value is FALSE.

copyId (optional): Indicates which copy to be retrieved. The default value is 0, which refers to the primary copy. Do not use this argument if you use the startByte and endByte arguments.

Note: The filename argument is required. All other arguments are optional except as noted.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

newFileName: The name of the retrieved file.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Retrieve specific file data from storage media. -->
<COMMAND name="FileRetrieve">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- fileName : name of file to retrieve -->
  <!-- newFileName : name of new file to retrieve to (default is "") -->
  <!-- startByte : start byte for partial retrieves (default is 0) -->
  <!-- endByte : end byte for partial retrieves (default is 0) -->
  <!-- modAccessTime : true, false (default is false) -->
  <!-- copyId : copy to retrieve (default is 0) -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The fileName argument is required. All others are optional. -->
  <!-- All arguments may be specified only once. -->
  <!--The copyId argument may not be specified with the startByte and -->
  <!-- endByte arguments. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="fileName" value="/csofs/storage/temp"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="FileRetrieve" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
<RETRIEVEINFO name="filename" value="/csofs/storage/temp"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful" />
</RESPONSE>
(The XML response is identical whether you use startByte and
endByte or copyId.)
```

C++ Class Declaration

```
class FileRetrieve : public Request
{
public:
   /// First Constructor
   /// This constructor is intended for instantiation of the
   object for a
   /// full retrieve.
   FileRetrieve(const std::string& inFileName);
```

```
/// Second Constructor
/// This constructor is intended for instantiation of the
object for a
/// full retrieve of the specified filename into the new
filename.
FileRetrieve(const std::string& inFileName,
    const std::string& inNewFileName);

/// Third Constructor
/// This constructor is intended for instantiation of the
object for a
/// partial retrieve of the specified filename into the new
filename
/// with startbyte and endbyte specifying the start/end of the
retrieve.
FileRetrieve(const std::string& inFileName,
    const std::string& inNewFileName,
    const int64_t& inStartByte,
    const int64_t& inEndByte);

/// Fourth Constructor
/// This constructor is intended for instantiation of the
object for a
/// full retrieve but with the copyId specified.
FileRetrieve(const std::string& inFileName,
    const int32_t& inCopyId);

/// Fifth Constructor
/// This constructor is intended for instantiation of the
object for a
/// full retrieve of the specified filename into the new
filename but
/// with the copyId specified.
FileRetrieve(const std::string& inFileName,
    const std::string& inNewFileName,
    const int32_t& inCopyId);

/// There are no set functions for copyid and startbyte/endbyte
/// because they are mutually exclusive options for this API.
/// These are handled by the various constructors defined
above.

/// Set Function for filename
void setFileName(std::string& inFileName);
```

```
    /// Set Function for modAccessTime
    void setModAccessTime(bool& inModAccessTime);

    /// Set Function for newfilename
    void setNewFileName(std::string& inNewFileName);

    /// Get Function for fileName
    const std::string& getFileName() const;

    /// Get Function for modAccessTime
    const bool& getModAccessTime() const;

    /// Get Function for copyId
    const int32_t& getCopyId() const;

    /// Get Function for newFileName
    const std::string& getNewFileName() const;

    /// Get Function for startByte
    const int64_t& getStartByte() const;

    /// Get Function for endByte
    const int64_t& getEndByte() const;

    /// Method to return the local status pair
    const StatusPair& getLocalStatus() const;
}
```

GetArchiveCapacity

This API provides the amount of remaining storage capacity for all archives.

Input

NA. This API has no command arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns the remaining storage capacity of all archives. -->
<COMMAND name="GetArchiveCapacity"/>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetArchiveCapacity" statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
  <STRINGINFO statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
    <!-- remainingMediaCapacity: (in GB) -->
    <INFO name="remainingMediaCapacity" value="99999"/>
 </STRINGINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetArchiveCapacity : public Request
{
public:
   /// Default and Primary Constructor
   GetArchiveCapacity();

   /// Method to return the total remaining capacity in GB.
   int32_t getCapacity() const;

   /// Method to return the local status
   const Status& getLocalStatus() const;
}
```

GetArchiveList

This API provides the current configuration settings for all archives.

Input

NA. This API has no command arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

archiveName: The name of the archive whose configuration settings are listed.

state: The current state of the archive: ONLINE or OFFLINE.

archiveType: The disk type for the archive (e.g., SCSI).

serialNumber: The archive's serial number.

model: The archive's model.

numberSlots: The total number of slots in the archive.

numberSlotsUsed: The number of used slots in the archive.

firmwareVersion: The archive's firmware version.

totalSpace: The amount of total space in the archive, in gigabytes.

remainingSpace: The amount of space, in gigabytes, remaining in the archive.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns configuration settings for all archives. -->
<COMMAND name="GetArchiveList"/>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetArchiveList" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <ARCHIVEINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
     <!-- state: {online, offline} -->
     <!-- totalSpace: (in GB) -->
     <!-- remainingSpace: (in GB) -->
     <INFO name="archiveName" value="Andromeda"/>
     <INFO name="state" value="online"/>
     <INFO name="archiveType" value="SCSI"/>
     <INFO name="serialNumber" value="ADIC_1_9Y2230000"/>
```

```
    <INFO name="model" value="Scalar 24"/>
    <INFO name="numberSlots" value="24"/>
    <INFO name="numberSlotsUsed" value="18"/>
    <INFO name="firmwareVersion" value="237A"/>
    <INFO name="totalSpace" value="93504"/>
    <INFO name="remainingSpace" value="90000"/>
  </ARCHIVEINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetArchiveList : public Request
{
public:
   /// Default and Primary Constructor
   GetArchiveList();

   /// Method to return the list of Archives
   const std::vector<ArchiveInfo>& getArchiveInfo() const;

   /// Method to return the local status
   const Status& getLocalStatus() const;
}

class ArchiveInfo : virtual public Info
{
public:
   /// Default and Primary Constructor
   ArchiveInfo(const std::string& inName="unknown",
        const std::string& inState="unknown",
        const std::string& inArchiveType="unknown",
        const std::string& inSerialNumber="unknown",
        const std::string& inModel="unknown",
        const int32_t&     inNumberSlots=0,
        const int32_t&     inNumberSlotsUsed=0,
        const std::string& inFirmwareVersion="unknown",
        const int32_t&     inTotalSpace=0,
        const int32_t&     inRemainingCapacity=0);

   /// Method to return the name of the archive.
   const std::string& getName() const;

   /// Method to return the state of the archive.
```

```
const std::string& getState() const;

/// Method to return the type of the archive.
const std::string& getArchiveType() const;

/// Method to return the serialnumber of the archive.
const std::string& getSerialNumber() const;

/// Method to return the model of the archive.
const std::string& getModel() const;

/// Method to return the number of slots for media in the
archive.
int32_t getNumberOfSlots() const;

/// Method to return the number of slots currently holding
media in the
/// archive.
int32_t getNumberOfSlotsUsed() const;

/// Method to return the firmware version of the archive.
const std::string& getFirmwareVersion() const;

/// Method to return the total capacity of the archive in
gigabytes.
int32_t getTotalSpace() const;

/// Method to return the total available (unused) capacity of
the
/// archive in gigabytes.
int32_t getRemainingSpace() const;

/// Method to set the state of the archive.
void setState(const std::string& inState);

/// Method to set the type of the archive.
void setArchiveType(const std::string& inArchiveType);

/// Method to set the serialnumber of the archive.
void setSerialNumber(const std::string& inSerialNumber);

/// Method to set the model of the archive.
void setModel(const std::string& inModel);
```

```
    /// Method to set the number of slots for media in the archive.
    void setNumberOfSlots(const int32_t& inNumberSlots);

    /// Method to set the number of slots currently holding media
    in
    /// the archive.
    void setNumberOfSlotsUsed(const int32_t& inNumberSlotsUsed);

    /// Method to set the firmware version of the archive.
    void setFirmwareVersion(const std::string& inFirmwareVersion);

    /// Method to set the total capacity of the archive in
    gigabytes.
    void setTotalSpace(const int32_t& inTotalSpace);

    /// Method to set the total available (unused) capacity of the
    /// archive in gigabytes.
    void setRemainingSpace(const int32_t& inRemainingSpace);

    /// Method to deflate this object for client-server transmision
    std::string deflate() const;

    /// Method to restore this object from a deflated string
    void inflate(std::string& inDeflatedObject);
}
```

Note: Currently, the GetArchiveList API assumes there is only one media type per archive. If the archive supports more than one media type, the output might not be accurate.

GetBackupStatus

This API provides the current status or progress of a backup in progress.

Input

NA. This API has no command arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

info: Any relevant information about the backup operation.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns the progression status of currently running backup
operation. -->
<COMMAND name="GetBackupStatus"/>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetBackupStatus" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <STRINGINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <!-- An INFO tag will surround each output line. -->
    <INFO value="Response text goes here."/>
    <INFO value="Response text goes here."/>
    <INFO value="Response text goes here."/>
    <INFO value="Response text goes here."/>
  </STRINGINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetBackupStatus : public Request
{
public:
   /// Default and Primary Contructor
   GetBackupStatus();

   /// Method to return the backup progression status
   const std::string& getBackupProgress() const;

   /// Method to return the local status
   const Status& getLocalStatus() const;
}
```

GetDriveList

This API provides a list of available drives and their attributes.

Input

NA. This API has no command arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

driveName: The name of the drive whose attributes are listed.

state: The drive's current state: ONLINE or OFFLINE.

serialNumber: The drive's serial number.

type: The type of drive.

mountState: The drive's current mount state: MOUNTED or UNMOUNTED.

mediaID: A media ID of the drive.

firmwareVersion: The drive's firmware version.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns a list of drives and their attributes. -->
<COMMAND name="GetDriveList"/>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetDriveList" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
 <!-- The DriveInfo will be repeated to specify multiple drives -->
 <DRIVEINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <!-- state: {online, offline} -->
  <!-- mountState: {mounted, unmounted} -->
  <!-- mediaID: a media ID or empty string -->
  <INFO name="driveName" value="Andromeda_LTO_Drive1"/>
```

```
   <INFO name="state" value="online"/>
   <INFO name="serialNumber" value="1110236593"/>
   <INFO name="type" value="LTO"/>
   <INFO name="mountState" value="mounted"/>
   <INFO name="mediaID" value="04132"/>
   <INFO name="firmwareVersion" value="4AP0"/>
  </DRIVEINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetDriveList : public Request
{
public:
   /// Default Contructor
   GetDriveList();

   /// Method to return the drive information
   const std::vector<DriveInfo>& getDriveInfo() const;

   /// Method to return the local status
   const Status& getLocalStatus() const;
}

class DriveInfo : virtual public Info
{
public:
   /// Primary Constructor
   DriveInfo(const std::string& inName="",
        const std::string& inState="",
        const std::string& inSerialNumber="",
        const std::string& inType="",
        const std::string& inMountState="",
        const std::string& inMediaID="",
        const std::string& inFirmwareVersion="");

   /// Method to retrieve the name of the drive
   const std::string& getName() const;

   /// Method to retrieve the state of the drive
   const std::string& getState() const;

   /// Method to retrieve the serial number of the drive
```

```
    const std::string& getSerialNumber() const;

    /// Method to retrieve the type of the drive
    const std::string& getType() const;

    /// Method to retrieve the mount-state of the drive
    const std::string& getMountState() const;

    /// Method to retrieve the media ID the drive contains, if any.
    const std::string& getMediaID() const;

   /// Method to retrieve the firmware version currently loaded on
   the drive
    const std::string& getFirmwareVersion() const;

    /// Method to deflate this object for client-server transmision
    std::string deflate() const;

    /// Method to restore this object from a deflated string
    void inflate(std::string& inDeflatedObject);
}
```

GetFileAttribute

This API returns attribute information for the specified file, which is located in either primary storage or secondary storage (tape or storage disk). See the Output section for complete details on the information retrieved.

Input

filename: The file's pathname.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

filename: The file name.

location: DISK, TAPE, or DISK AND TAPE.

policyClass: The name of the policy class associated with the specified file.

numberExistingCopies: The number of existing copies. Valid values are 1 to [*numberTargetCopies*].

numberTargetCopies: The number of total copies.

mediaID: The media ID where the copy resides. There could be a list of media IDs.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns the attributes for a specific file. -->
<COMMAND name="GetFileAttribute">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- fileName : name of file -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The fileName argument is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="fileName" value="/snfs/myDirectory/myFile.dat"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetFileAttribute" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <FILEINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
     <!-- location: {DISK, TAPE, DISK AND TAPE} -->
     <INFO name="filename" value="/snfs/myDirectory/myFile.dat"/>
     <INFO name="location" value="TAPE"/>
     <INFO name="policyClass" value="policyclass1"/>
     <INFO name="numberExistingCopies" value="1"/>
     <INFO name="numberTargetCopies" value="1"/>
     <INFO name="mediaID" value="000001"/>
  </FILEINFO>
</RESPONSE>
```

C++ Class Declaration:

```
class GetFileAttribute : public Request
{
public:
```

```
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
the object,
   /// providing it the filename
   GetFileAttribute(const std::string& filename);

   /// Copy constructor
   GetFileAttribute(const GetFileAttribute& obj);

   /// Destructor
   virtual ~GetFileAttribute();

   /// Assignment Operator
   GetFileAttribute& operator=(const GetFileAttribute& obj);

   /// Method to reset object with new filename
   void reset(const std::string& filename);

   /// Method to return the file information
   const FileInfo& getFileInfo() const;

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}


class FileInfo : virtual public Info
{
public:
   /// Location values
   enum Location
   {
      unknown=0,
      disk,
      tape,
      diskandtape,
   };

   /// Default and Primary Constructor
   /// This can construct from a single mediaID string.
   FileInfo(const std::string& inName="",
       const Location&    inLocation=unknown,
       const std::string& inPolicyClass="",
```

```
        const int32_t&     inNumberOfExistingCopies=0,
        const int32_t&     inNumberOfTargetCopies=0,
        const std::string& inMediaID="");

    /// Secondary Constructor
    /// This constructs from a vector of mediaID.
    FileInfo(const std::string& inName,
        const Location&    inLocation,
        const std::string& inPolicyClass,
        const int32_t&     inNumberOfExistingCopies,
        const int32_t&     inNumberOfTargetCopies,
        const MediaList&   inMedia);

    /// Method that retrieves the file name.
    const std::string& getFileName() const;

    /// Method that retrieves the file location.
    const Location& getLocation() const;

    /// Method that retrieves the file location (as string).
    const std::string& getLocationAsString() const;

    /// Method that retrieves the file's policy class.
    const std::string& getPolicyClass() const;

    /// Method that retrieves the number of existing copies of the
    file.
    int32_t getNumberOfExistingCopies() const;

    /// Method that retrieves the number of target copies of the
    file.
    int32_t getNumberOfTargetCopies() const;

    /// Method that retrieves the collection of media IDs.
    const MediaList& getMedia() const;

    /// Method to deflate this object for client-server transmision
    std::string deflate() const;

    /// Method to restore this object from a deflated string
    void inflate(std::string& inDeflatedObject);
}
```

GetFileTapeLocation

This API returns location information for the specified file, including the copy ID, segment number, starting block, and segment size. See the Output section for complete details about the information returned.

Input

filename: The file's pathname.

copyID: Indicates which copy from which to get segment information. The default value is 1, the primary copy.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

filename: The file name.

copyID: The copy ID.

segment: The segment number. (There can be multiple segments.)

mediaID: The media ID where this segment resides.

archiveID: The archive ID where the media belongs.

startBlock: The starting block.

offset: The offset from startBlock.

segmentSize: The segment size.

blockSize: The media block size.

Note: The file's data can be located by *startBlock* and *offset*. Then *segsize* will tell the amount of data to be read. *blksize* is required for doing a correct read operation; otherwise, enough data might not be read when reading a block. Both StorNext and AMASS tape formats are supported by this API.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Return the tape location info for a specific file. -->
<COMMAND name="GetFileTapeLocation">
 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
 <!-- fileName : name of file -->
 <!-- copyID : copy number -->
 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
 <!-- Exactly one each of the above arguments are required. -->
 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
 <ARGUMENT name="fileName" value="/snfs/myDirectory/myFile.dat"/>
 <ARGUMENT name="copyID" value="1"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetFileTapeLocation" statusCode="0"
status="SUCCESS"
       statusDescription="Command Successful">
 <LOCATIONINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <INFO name="fileName" value="/stornnnnext/snfs1/d1/regfile" />
   <INFO name="copyID" value="1" />
   <SEGMENTINFO name="segmentNumber" value="1">
     <INFO name="mediaID" value="000455" />
     <INFO name="archiveID" value="scsi_archive1" />
     <INFO name="startBlock" value="7" />
     <INFO name="offset" value="128" />
     <INFO name="segmentSize" value="100000" />
     <INFO name="blockSize" value="524288" />
   </SEGMENTINFO>
   <SEGMENTINFO name="segmentNumber" value="2">
     <INFO name="mediaID" value="000455" />
     <INFO name="archiveID" value="scsi_archive1" />
     <INFO name="startBlock" value="7" />
     <INFO name="offset" value="100256" />
     <INFO name="segmentSize" value="100000" />
     <INFO name="blockSize" value="524288" />
   </SEGMENTINFO>
   <SEGMENTINFO name="segmentNumber" value="3">
```

```
      <INFO name="mediaID" value="000455" />
      <INFO name="archiveID" value="scsi_archive1" />
      <INFO name="startBlock" value="7" />
      <INFO name="offset" value="200384" />
      <INFO name="segmentSize" value="100000" />
      <INFO name="blockSize" value="524288" />
   </SEGMENTINFO>
 </LOCATIONINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetFileTapeLocation : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
the object,
   /// providing it the filename
   GetFileTapeLocation(const std::string& filename,
        const int16_t&     copyid = 1);

   /// Method to set the copy id.
   void setCopyID(const int16_t inCopyID);

   /// Method to return the location info
   const LocationInfo& getLocationInfo() const;

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}

class LocationInfo : virtual public Info
{
public:
   /// Primary Constructor
   LocationInfo(const std::string& inName="",
        const int16_t&     inCopyID=1);

   /// Method that set the file name
   void setFileName(const std::string& filename);

   /// Method that set the copy id
```

```
    void setCopyID(const int16_t& copyid);

    /// Method that retrieves the file name.
    const std::string& getFileName() const;

    /// Method that retrieves the file location.
    const int16_t& getCopyID() const;

    /// Method that add a segment info to the segment list
    void addSegment(const SegmentInfo& seginfo);

    /// Method that retrieves the collection of segment locations
    const std::vector<SegmentInfo>& getSegmentList() const;

    /// Method to deflate this object for client-server transmision
    std::string deflate() const;

    /// Method to restore this object from a deflated string
    void inflate(std::string& inDeflatedObject);
}
```

GetMediaList

This API returns either a list of media IDs or the number of media that meets the specified criteria.

Input arguments within brackets {} must be entered explicitly as shown. For example, when entering the argument for the format argument, you must enter “count” or “list” (without the quotation marks).

All arguments are optional. If you do not specify any arguments, the total number of media is returned.

Input

format: {count or list}

location: {archive, vault, checkout, or unknown} If no argument is specified, archive is used as the default value.

archiveID: The archive ID.

classification: {data, backup, or cleaning}

availability: {available or unavailable}

writeAccess: {writeProtected or notWriteProtected}

integrity: {suspect or notSuspect}

space: {full or blank}

percentUsed: The percentage of media space used, from 0.00 to 100.00.

copyID: The number of the copy used, from 1 - 8.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

location: archive, vault, checkout, or unknown.

archiveID: The archive ID.

classification: data, backup, or cleaning.

availability: available or unavailable.

writeAccess: writeProtected or notWriteProtected.

integrity: suspect or notSuspect.

space: full or blank.

count: The number of media that meet the specified criteria.

medialist: A list of media that meet the specified criteria.

Note: Output will include either count or list (not both), depending on what you specified as an input argument. Output will also repeat the selection criteria specified by the input.

XML Example

The following example illustrates how to return the number of blank media for a given archive.

Request Example 1:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Return a list of media based on specific query criteria. -->
<COMMAND name="GetMediaList">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- format : count, list (default is count) -->
```

```
  <!-- location : archive, vault, checkout, unknown -->
  <!-- archiveID : archive name -->
  <!-- classification : data, backup, cleaning -->
  <!-- availability : available, unavailable -->
  <!-- writeAccess : writeProtected, notWriteProtected -->
  <!-- integrity : suspect, notSuspect -->
  <!-- space : blank, partial, full -->
  <!-- percentUsed : 0.00 ... 100.00 -->
  <!-- copyID : 1 ... 8 -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- All arguments are optional and may be specified only once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="format" value="count"/>
  <ARGUMENT name="location" value="archive"/>
  <ARGUMENT name="availability" value="available"/>
</COMMAND>
```

Response Example 1:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetMediaList" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <MEDIALIST statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <INFO name="location" value="archive"/>
    <INFO name="availability" value="available"/>
    <INFO name="count" value="6"/>
  </MEDIALIST>
</RESPONSE>
```

Request Example 2:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<COMMAND name="GetMediaList">
  <ARGUMENT name="format" value="list"/>
  <ARGUMENT name="location" value="archive"/>
  <ARGUMENT name="availability" value="available"/>
</COMMAND>
```

Response Example 2:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetMediaList" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <MEDIALIST statusCode="0" status="SUCCESS"
```

```
       statusDescription="Command Successful">
     <INFO name="location" value="archive"/>
     <INFO name="availability" value="available"/>
     <INFO name="mediaID" value="000001"/>
     <INFO name="mediaID" value="000002"/>
     <INFO name="mediaID" value="000003"/>
     <INFO name="mediaID" value="000004"/>
     <INFO name="mediaID" value="000005"/>
     <INFO name="mediaID" value="000006"/>
   </MEDIALIST>
</RESPONSE>
```

Request Example 3:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<COMMAND name="GetMediaList">
  <ARGUMENT name="archiveID" value="scsi_archive1"/>
  <ARGUMENT name="space" value="blank"/>
</COMMAND>
```

Response Example 3:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetMediaList" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <MEDIALIST statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
     <INFO name="archiveID" value="scsi_archive1" />
     <INFO name="space" value="blank" />
     <INFO name="count" value="3" />
   </MEDIALIST>
</RESPONSE>
```

C++ Class Declaration

```
class GetMediaList : public Request
{
public:
   /// Output format values
   enum Format { count=0, list };

   /// Location values used as selection criteria
   enum Location { archive=0, vault, checkout, unknown };
```

```
/// Classification values used as selection criteria
enum Classification { data=0, backup, cleaning };

/// Availability values used as selection criteria
enum Availability { available=0, unavailable };

/// Write access values used as selection criteria
enum WriteAccess { writeProtected=0, notWriteProtected };

/// Integrity values used as selection criteria
enum Integrity { suspect=0, notSuspect };

/// Space values used as selection criteria
enum Space { blank=0, partial, full };

/// Default Constructor (format defaults to count)
GetMediaList();

/// Primary Constructor
/// Note: This constructor will be deprecated in a future
release.
GetMediaList(const Format&          inFormat,
    const Location&        inLocation,
    const std::string&     inArchiveID,
    const Classification&  inClassification,
    const Availability&    inAvailability,
    const WriteAccess&     inWriteAccess,
    const Integrity&       inIntegrity,
    const Space&           inSpace);

/// Set the output format for the media list.
void setFormat(const Format& inFormat);

/// Set the location for the media selection criteria.
void setLocation(const Location& inLocation);

/// Set the archive ID for the media selection criteria.
void setArchiveID(const std::string& inArchiveID);

/// Set the classification for the media selection criteria.
void setClassification(const Classification& inClassification);

/// Set the availability for the media selection criteria.
```

```
void setAvailability(const Availability& inAvailability);

/// Set the write access for the media selection criteria.
void setWriteAccess(const WriteAccess& inWriteAccess);

/// Set the integrity for the media selection criteria.
void setIntegrity(const Integrity& inIntegrity);

/// Set the space for the media selection criteria.
void setSpace(const Space& inSpace);

/// Set the percentUsed for the media selection criteria.
void setPercentUsed(const float inPercentUsed);

/// Set the copyID for the media selection criteria.
void setCopyID(const int32_t inCopyID);

/// Return the output format specified for the media list.
const Format& getFormat() const;

/// Return the media location selection criteria
const Location& getLocation() const;

/// Return the media archive ID selection criteria
const std::string& getArchiveID() const;

/// Return the media classification selection criteria
const Classification& getClassification() const;

/// Return the media availability selection criteria
const Availability& getAvailability() const;

/// Return the media write access selection criteria
const WriteAccess& getWriteAccess() const;

/// Return the media integrity selection criteria
const Integrity& getIntegrity() const;

/// Return the media space selection criteria
const Space& getSpace() const;

/// Return the media percentUsed selection criteria
const float getPercentUsed() const;
```

```
    /// Return the media copyID selection criteria
    const int32_t getCopyID() const;

    /// Return the media list output format as a string.
    const std::string& getFormatAsString() const;

    /// Return the media location selection criteria as a string
    type.
    const std::string& getLocationAsString() const;

    /// Return the media archive id selection criteria as a string
    type.
    const std::string& getArchiveIDAsString() const;

    /// Return the media classification selection criteria as a
    string type.
    const std::string& getClassificationAsString() const;

    /// Return the media availability selection criteria as a
    string type.
    const std::string& getAvailabilityAsString() const;

    /// Return the media write access selection criteria as a
    string type.
    const std::string& getWriteAccessAsString() const;

    /// Return the media integrity selection criteria as a string
    type.
    const std::string& getIntegrityAsString() const;

    /// Return the media space selection criteria as a string type.
    const std::string& getSpaceAsString() const;

    /// Method to return the count of Media
    int32_t getMediaCount() const;

    /// Method to return the list of Media
    const std::vector<std::string>& getMediaList() const;

    /// Method to return the local status
    const Status& getLocalStatus() const;
}
```

GetMediaStatus

This API returns status for the specified piece of media. See the Output section for complete details on the information retrieved.

Input

mediaID: The media ID. There could be a list of media IDs.

Output

location: The current state for the specified media ID: CHECKOUT, INTRANSIT, or UNKNOWN

writeProtected: Y or N.

status: AVAIL or UNAVAIL.

spaceUsed: The amount of used space in bytes.

spaceFree: The amount of free space in bytes.

classification: Media classification: DATA, BACKUP, or CLEANING.

mediaID: The media ID.

policyClass: The policy class associated with the media.

location: Archive name.

type: The media type.

writeProtected: Y or N.

mountCount: The number of mounts.

status: AVAIL or UNAVAIL.

suspectCount: The total number of times media was marked as suspect.

spaceUsed: The amount of used space in bytes.

spaceFree: The amount of free space in bytes.

percentUsed: The percentage of space used.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns status info on all specified media. -->
```

```
  <COMMAND name="GetMediaStatus">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- mediaID : valid media ID -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- One or more mediaID must be specified. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="mediaID" value="025311"/>
  <ARGUMENT name="mediaID" value="025312"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetMediaStatus" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <MEDIAINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <!-- location: {<archiveName>, checkout, intransit, unknown} -->
    <!-- writeProtected: {Y, N} -->
    <!-- status: {AVAIL, UNAVAIL} -->
    <!-- spaceUsed: (in bytes) -->
    <!-- spaceFree: (in bytes) -->
    <!-- classification: {data, backup, cleaning} -->
    <INFO name="mediaID" value="025311"/>
    <INFO name="policyClass" value="policyclass1"/>
    <INFO name="location" value="Andromeda"/>
    <INFO name="type" value="LTO"/>
    <INFO name="writeProtected" value="N"/>
    <INFO name="mountCount" value="13"/>
    <INFO name="status" value="AVAIL"/>
    <INFO name="suspectCount" value="0"/>
    <INFO name="spaceUsed" value="6,442,451,356"/>
    <INFO name="spaceFree" value="191,973,294,080"/>
    <INFO name="percentUsed" value="3.25"/>
    <INFO name="
  </MEDIAINFO>
</RESPONSE>
```

C++ Class Declaration:

```
class GetMediaStatus : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
  the object,
   /// providing it the mediaId
   GetMediaStatus(const std::string& inMediaID);

   /// Constructor from a list of media
   /// This constructor is intended to create the request from a
  list of media
   GetMediaStatus(const MediaList& inMedia);

   /// Method to return a MediaInfo for a specific media
   MediaInfo getMediaInfo(const std::string& inMediaID);

   /// Method to return list of media info
   /// The media info list contains only the good media results.
   const std::vector<MediaInfo>& getMediaInfo() const;

   /// Method to return the list of status pairs
   const StatusPairList& getLocalStatus() const;

   /// Method to return a specific status pair for a given mediaID
   const StatusPair& getLocalStatus(const std::string& mediaID)
   const;
}

class MediaInfo : virtual public Info
{
public:
   /// Default and Primary Constructor
   MediaInfo(const std::string& inMediaID="unknown",
        const std::string& inPolicyClass="unknown",
        const std::string& inLocation="unknown",
        const std::string& inType="unknown",
        const std::string& inWriteProtected="unknown",
        const uint32_t     inMountCount=0,
        const std::string& inStatus="unknown",
        const uint32_t     inSuspectCount=0,
        const std::string& lastAccessTime="unknown",
```

```
    const uint64_t     inSpaceUsed=0,
    const uint64_t     inSpaceFree=0,
    const float        inPercentUsed=0,
    const std::string& inClassification="unknown");

/// Method to return the media ID
const std::string& getMediaID() const;

/// Method to return the media policy class
const std::string& getPolicyClass() const;

/// Method to return the media location
const std::string& getLocation() const;

/// Method to return the media type
const std::string& getType() const;

/// Method to return flag indicating if the media is write-
protected
const std::string& getWriteProtected() const;

/// Method to return the mount count for the media
const uint32_t getMountCount() const;

/// Method to return the media status
const std::string& getStatus() const;

/// Method to return the total number of times the media was
marked
/// suspect due to errors.
const uint32_t getSuspectCount() const;

/// Method to return the media last access time
const std::string& getLastAccessTime() const;

/// Method to return the amount of space used on the media
const uint64_t getSpaceUsed() const;

/// Method to return the amount of free space on the media
const uint64_t getSpaceFree() const;

/// Method to return the percentage of space used on the media
const float   getPercentUsed() const;
```

```
   /// Method to return the classification of media (backup, data,
   clean, etc.)
    const std::string& getClassification() const;

    /// Method to deflate this object for client-server transmision
    std::string deflate() const;

    /// Method to restore this object from a deflated string
    void inflate(std::string& inDeflatedObject);
}
```

GetPolicy

This API retrieves the specified policy.

Input

policyClass: The desired policy class. Enter a valid policy class name or **all** to retrieve all policy classes.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

policyClass: The policy class name(s).

numberCopies: The number of copies currently maintained for the policy class.

maxInactiveVersions: The maximum number of versions currently maintained for the policy class.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns configuration info on one or all policy classes. -->
<COMMAND name="GetPolicy">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!- policyClass : valid policy class name or "all" (default is all) ->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
```

```
  <!-- The policyClass argument is optional and may be specified -->
  <!-- only once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="policyClass" value="mypolicyclass"/>
</COMMAND>
```

Response:


```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetPolicy" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <POLICYINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <INFO name="policyClass" value="mypolicyclass"/>
    <INFO name="numberCopies" value="2"/>
    <INFO name="maxInactiveVersions" value="10"/>
  </POLICYINFO>
</RESPONSE>
```


C++ Class Declaration

```
class GetPolicy : public Request
{
public:
   /// Default Contructor
   GetPolicy(const std::string& inPolicyClass="all");

   /// Method to return the list of PolicyInfo objects
   const std::vector<PolicyInfo>& getPolicyInfo() const;

   /// Method to return a specific PolicyInfo object
   const PolicyInfo& getPolicyInfo(const std::string&
   inPolicyClass) const;

   /// Method to return the list of status pairs
   const StatusPairList& getLocalStatus() const;

   /// Method to return a specific status pair for a given policy
   class
   const StatusPair& getLocalStatus(const std::string&
inPolicyClass) const;
}

class PolicyInfo : virtual public Info
```

```
{
public:
   /// Primary constructor
   PolicyInfo(const std::string& inClassName,
        const int32_t&     inNumberOfCopies,
        const int32_t&     inMaxInactiveVersions);

   /// Method to return the policy classname.
   const std::string& getPolicyClassName() const;

   /// Method to return the number of copies to make.
   const int32_t& getNumberOfCopies() const;

   /// Method to return the maximum inactive versions to keep.
   const int32_t& getMaxInactiveVersions() const;

   /// Method to deflate this object for client-server transmision
   std::string deflate() const;

   /// Method to restore this object from a deflated string
   void inflate(std::string& inDeflatedObject);
}
```

GetPortList

This API retrieves the import/export port IDs for a specified archive. An archive can have multiple ports. (Vaults do not have a port, so the ID is always 0.)

Input

archiveID: The ID of the archive whose import/export port information you want to retrieve.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

portID: The port number.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Return IE port information on a specific archive. -->
<COMMAND name="GetPortList">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- archiveID : valid archive name -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The archiveID argument is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="archiveID" value="scsi_archive1"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetPortList" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <!-- The PORTINFO will be repeated to specify multiple ports -->
  <PORTINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <INFO name="portID" value="16"/>
  </PORTINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetPortList : public Request
{
public:
   /// Primary Constructor
   GetPortList(const std::string& inArchiveId);

   /// Method to return the port IDs
   const std::vector<PortInfo>& getPortInfo() const;

   /// Method to return the local status
   const Status& getLocalStatus() const;
}

class PortInfo : virtual public Info
{
```

```
public:
   /// Default and Primary Constructor
   PortInfo(const int16_t& inPortID=0);

   /// Method to return the port ID for the archive.
   int16_t getPortID() const;

   /// Method to set the port ID for the archive
   void setPortID(const int16_t& inPortID);

   /// Method to deflate this object for client-server transmision
   std::string deflate() const;

   /// Method to restore this object from a deflated string
   void inflate(std::string& inDeflatedObject);
}
```

GetSchedule

This API provides information about previously scheduled events such as backups.

Input

scheduleType: BACKUP

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

scheduleType: The type of schedule (e.g., BACKUP)

scheduleTime: Time the scheduled event is set to run, in 24-hour format: HH:MM.

scheduleLastAttempt: Time the scheduled event last attempted to run.

scheduleLastStatus: Status of the last scheduled event attempt: SUCCESSFULL, RUNNING, LOCKED, TERMINATED, or FAILED.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Get the time for the default full backup and default -->
<!-- partial backup schedules. -->
<COMMAND name="GetSchedule">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- scheduleType : backup -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Each argument above is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="scheduleType" value="backup"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetSchedule" statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
  <SCHEDULEINFO statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
    <!-- scheduleTime: (in 24-hour HH:MM format) -->
    <!-- scheduleLastStatus: {Successful, Running, Locked,
    Terminated, Failed} -->
    <INFO name="scheduleType" value="backup"/>
    <INFO name="scheduleTime" value="23:30"/>
    <INFO name="scheduleLastAttempt" value="Jul 9,2011 00:30"/>
    <INFO name="scheduleLastStatus" value="Successful"/>
  </SCHEDULEINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetSchedule : public Request
{
public:
   /// Default and Primary Constructor
   GetSchedule(const std::string& inScheduleType="backup");

   /// Method to return the ScheduleInfo
   const ScheduleInfo& getScheduleInfo() const;
```

```
   /// Method to return the local status.
   const Status& getLocalStatus() const;
}

class ScheduleInfo : virtual public Info
{
public:
   /// State Values
   static const std::string unknown;

   /// Default and Primary Constructor
   ScheduleInfo(const std::string& inType=unknown,
        const std::string& inRunTime=unknown,
        const std::string& inLastAttempt=unknown,
        const std::string& inLastStatus=unknown,
        const std::string& inPeriod=unknown,
        const std::string& inDay=unknown);

   /// Method to return the type of the schedule.
   const std::string& getType() const;

   /// Method to return the run time of the schedule.
   const std::string& getRunTime() const;

   /// Method to return the last attempted run time of the
   schedule.
   const std::string& getLastAttempt() const;

   /// Method to return the status of last attempted run of the
   schedule.
   const std::string& getLastStatus() const;

   /// Method to return the period of the schedule.
   const std::string& getPeriod() const;

   /// Method to return the day of the schedule.
   const std::string& getDay() const;

   /// Method to set the type of the schedule.
   void setType(const std::string& inType);

   /// Method to set the run time of the schedule.
   void setRunTime(const std::string& inRunTime);
```

```
    /// Method to set the last attempted run time of the schedule.
    void setLastAttempt(const std::string& inLastAttempt);

    /// Method to set the status of last attempted run of the
    schedule.
    void setLastStatus(const std::string& inLastStatus);

    /// Method to set period (daily/weekly/monthly) the of the
    schedule.
    void setPeriod(const std::string& inPeriod);

    /// Method to set the day of the schedule.
    void setDay(const std::string& inWeekDay);

    /// Method to deflate this object for client-server transmision
    std::string deflate() const;

    /// Method to restore this object from a deflated string
    void inflate(std::string& inDeflatedObject);
}
```

GetSystemStatus

This API provides the current system status.

Input

NA. This API has no command arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

version: The current StorNext software version number.

systemState: The overall system status.

tsm: The current TSM state (ONLINE or OFFLINE).

msm: The current MSM state (ONLINE or OFFLINE).

dsm: The current DSM state (ONLINE or OFFLINE).

database: The current database state (ONLINE or OFFLINE).

svclog: The current service log state (ONLINE or OFFLINE).

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Returns the current system status. -->
<COMMAND name="GetSystemStatus"/>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="GetSystemStatus" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <SYSTEMINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <INFO name="version" value="2.7.0(18)"/>
    <!-- The systemState is the overall system status -->
    <!-- The values below will be "online" or "offline" -->
    <INFO name="systemState" value="online"/>
    <INFO name="tsm" value="online"/>
    <INFO name="msm" value="online"/>
    <INFO name="dsm" value="online"/>
    <INFO name="database" value="online"/>
    <INFO name="svclog" value="online"/>
  </SYSTEMINFO>
</RESPONSE>
```

C++ Class Declaration

```
class GetSystemStatus : public Request
{
public:
   /// Default and Primary Constructor
   GetSystemStatus();

   /// Method to return the system information
   const SystemInfo& getSystemInfo() const;

   /// Method to return the local status
   const Status& getLocalStatus() const;
}

class SystemInfo : virtual public Info
```

```
{
public:
   /// Component Values
   enum Component
   {
      system=0,
      database,
      dsm,
      msm,
      svclog,
      tsm,
   };

   /// State Values
   static const std::string unknown;
   static const std::string online;
   static const std::string offline;

   /// Default and Primary Constructor
   SystemInfo(const std::string& inSystemVersion=unknown,
       const std::string& inSystemState=unknown,
       const std::string& inTsmState=unknown,
       const std::string& inMsmState=unknown,
       const std::string& inDsmState=unknown,
       const std::string& inDatabaseState=unknown,
       const std::string& inSvclogState=unknown);

   /// Method to return the version of the system
   const std::string& getSystemVersion() const;

   /// Method to return the state of the specified component
   const std::string& getState(const Component& inComponent)
   const;

   /// Method to set the system version.
   void setSystemVersion(const std::string& inSystemVersion);

   /// Method to set the state of the specified component.
   void setState(const Component& inComponent,
                 const std::string& inState);

   /// Method to deflate this object for client-server transmision
   std::string deflate() const;
```

```
   /// Method to restore this object from a deflated string
   void inflate(std::string& inDeflatedObject);
}
```

MoveMedia

This API logically marks a media to move from one archive to another. It does not do the physical moving. In order to both physically and logically move a media between two archives, you must take three steps:

1. Call MoveMedia.
2. Call EjectMedia to physically move the media to the mailbox at the source archive.
3. Call EnterMedia to physically move the media from mailbox into the destination archive.

Note: If the media is entered or ejected from a vault, EjectMedia and EnterMedia are logical operations.

Input

mediaID: The media ID to move.

destArchiveID: The destination archive ID to move to.

Note: Both of these arguments are required.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Move a specific media to another archive. -->
<COMMAND name="MoveMedia">
```

```
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- mediaID : media to move -->
  <!-- destArchiveID : destination archive name -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Exactly one each of the above arguments are required. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="mediaID" value="000455"/>
  <ARGUMENT name="destArchiveID" value="vault1"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="MoveMedia" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <STATUSDETAIL name="mediaID" value="000455"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class MoveMedia : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// given a destination archive ID and a single media ID.
   MoveMedia(const std::string& inMediaID,
        const std::string& inDestArchiveID);

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}
```

PassThru

This API executes the specified command line argument. The PassThru API supports running only one CLI command at a time. You cannot string together multiple concatenated commands (e.g. ls /tmp/; ls /var/tmp/; cd /tmp/).

Input

commandString: The command you want to execute from the command line (e.g., ls -laF /tmp /)

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

Any response for the executed command.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Executes a specific shell command. -->
<COMMAND name="PassThru">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- commandString : valid shell command -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The commandString argument is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="commandString" value="ls -laF /tmp"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="PassThru" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <STRINGINFO statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
    <!-- An INFO tag will surround each output line. -->
    <INFO value="Response text goes here."/>
    <INFO value="Response text goes here."/>
    <INFO value="Response text goes here."/>
    <INFO value="Response text goes here."/>
  </STRINGINFO>
</RESPONSE>
```

C++ Class Declaration

```
class PassThru : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
  the object,
   /// given a command string.
   PassThru(const std::string& inCommand);

   /// Method to return the passthru command string
   const std::string& getCommandString() const;

   /// Method to return the results of the passthru command
   const std::string& getCommandResults() const;

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}
```

RmDiskCopy

This API removes the disk copy of the specified file through explicit truncation.

Input

filename: The pathname of the file whose on-disk copy will be removed.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Truncate file data from disk. -->
<COMMAND name="RmDiskCopy">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
```

```
  <!-- fileName : name of file -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The fileName argument is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="fileName" value="/snfs/myDirectory/myFile.dat"/
>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="RmDiskCopy" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
   <STATUSDETAIL name="filename" value="/snfs/myDirectory/
   myFile.dat"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class RmDiskCopy : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
the object,
   /// given a single filename.
   RmDiskCopy(const std::string& filename);

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}
```

SetArchiveState

This API allows you to set the archive state to ON or OFF.

Input

archiveName: The name of the archive for which you want to set the state.

state: ON or OFF.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Set the operating state of a specific archive. -->
<COMMAND name="SetArchiveState">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- archiveName : archive name -->
  <!-- state : ON, OFF -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Each argument above is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="archiveName" value="Andromeda"/>
  <ARGUMENT name="state" value="ON"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetArchiveState" statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
    <STATUSDETAIL name="archiveName" value="Andromeda"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetArchiveState : public Request
{
public:
   /// State values
   enum StateType
   {
      ONLINE=0,
      OFFLINE
   };
```

```
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
  the
   /// object, providing it the archive name and state.
   SetArchiveState(const std::string& inArchiveName,
        const StateType&   inArchiveState);

   /// Method to return the archive name
   const std::string& getArchiveName() const;

   /// Method to return the archive state
   const StateType& getArchiveState() const;

   /// Method to return the archive state as a string
   const std::string& getArchiveStateAsString() const;

   /// method to return the local status
   const Status& getLocalStatus() const;
}
```

SetDirAttributes

This API allows you to set the following directory attributes:

- store (enable or disable)
- truncate (enable or disable)
- policy class name for the directory

Input

directoryName: The name of the directory for which you want to set attributes.

noTruncate: TRUE or FALSE.

noStore: TRUE or FALSE.

policyClass: The name of the policy class you want to apply to the directory.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Set the attributes for a specific directory. -->
<COMMAND name="SetDirAttributes">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- directoryName : name of directory -->
  <!-- noTruncate : TRUE, FALSE -->
  <!-- noStore : TRUE, FALSE -->
  <!-- policyClass : valid policy class name -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The directoryName argument is required exactly once. -->
  <!-- At least one of the following arguments are required, but -->
  <!-- no more than one of each: -->
  <!-- noTruncate -->
  <!-- noStore -->
  <!-- policyClass -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="directoryName" value="/snfs/mydirectory"/>
  <ARGUMENT name="noTruncate" value="TRUE"/>
  <ARGUMENT name="noStore" value="TRUE"/>
  <ARGUMENT name="policyClass" value="mypolicyclass"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetDirAttributes" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <STATUSDETAIL name="directoryName" value="/snfs/mydirectory"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetDirAttributes : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// providing it the directory name, policy class, truncate,
   and store
   /// settings.
   /// the state
   SetDirAttributes(const std::string& inDirectoryName,
       const std::string& inPolicyClass,
       const bool& inNoTruncate,
       const bool& inNoStore);

   /// Secondary Constructor
   /// This constructor is a secondary means to instantiate the
   object,
   /// the state
   SetDirAttributes(const std::string& inDirectoryName);

   /// Method to return the directory name
   const std::string& getDirectoryName() const;

   /// Method to return the policy class
   const std::string& getPolicyClass() const;

   /// Method to return the noTruncate flag
   bool getNoTruncateFlag() const;

   /// Method to return the noStore flag
   bool getNoStoreFlag() const;

   /// Method to return the local status
   const StatusPair& getLocalStatus() const;
}
```

SetDriveState

This API allows you to set the drive state to ON or OFF.

Input

drivename: The name of the drive whose state you want to set.

state: ON or OFF.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Set the operating state of a specific drive. -->
<COMMAND name="SetDriveState">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- drivename : valid drive name -->
  <!-- state : ON, OFF -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Exactly one each of the above arguments is required. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="drivename" value="lto2-001"/>
  <ARGUMENT name="state" value="ON"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetDriveState" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <STATUSDETAIL name="drivename" value="Andromeda_LTO_Drive1"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetDriveState : public Request
{
public:
   /// Drive state values.
   enum DriveState
   {
      ON = 0,
      OFF
   };

   /// Primary Constructor
   /// This constructor is intended for primary instantiation
   /// of the object, providing it the drive name and state.
   SetDriveState(const std::string& inDriveName,
       const DriveState&  inDriveState);

   /// Method to return the drivename
   std::string getDrivename() const;

   /// Method to return the drive state
   DriveState getDriveState() const;

   /// Method to return the drive state as a string
   std::string getDriveStateAsString() const;

   /// Method to return the local status pair
   const StatusPair& getLocalStatus() const;
}
```

SetFileAttributes

This API allows you to set the following attributes for a file:

- store (enable or disable)
- truncate (enable or disable)

Input

fileName: The name of the file for which you want to set attributes.

noStore: TRUE or FALSE.

noTruncate: TRUE or FALSE.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Set the attributes for a specific file. -->
<COMMAND name="SetFileAttributes">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- fileName : name of file -->
  <!-- noTruncate : TRUE, FALSE -->
  <!-- noStore : TRUE, FALSE -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The fileName argument is required exactly once. -->
  <!-- At least one of the following arguments are required, but -->
  <!-- no more than one of each: -->
  <!-- noTruncate -->
  <!-- noStore -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="fileName" value="/snfs/myDirectory/myFile.dat"/
>
  <ARGUMENT name="noTruncate" value="TRUE"/>
  <ARGUMENT name="noStore" value="TRUE"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetFileAttributes" statusCode="0"
status="SUCCESS"
       statusDescription="Command Successful">
  <STATUSDETAIL name="filename" value="/snfs/myDirectory/
  myFile.dat"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetFileAttributes : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// providing it the file name, notruncate, and nostore
   attributes to be
   /// set for the file.
   SetFileAttributes(const std::string& inFileName,
        const bool&        inNoTruncate,
        const bool&        inNoStore);

   /// Secondary Constructor
   /// This constructor is a secondary means to instantiate the
   object,
   SetFileAttributes(const std::string& inFileName);

   /// Method to return the fileName
   const std::string& getFileName() const;

   /// Method to return the noTruncate flag
   bool getNoTruncateFlag() const;

   /// Method to return the noStore flag
   bool getNoStoreFlag() const;

   /// Method to return the local status
   const StatusPair& getLocalStatus() const;
}
```

SetMediaState

This API allows you to set the state for one or more piece of media.

Input

state: AVAIL (available), UNAVAIL (not available), PROTECT (write protected), UNPROTECT (not write protected), UNMARK (not marked), or UNSUSP (not suspect).

mediaID: The ID of the media whose state you want to set. When entering multiple media IDs, enter one media ID per line.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code for each media ID entered.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Set the operating state for specific media. -->
<COMMAND name="SetMediaState">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- state : avail, unavail, protect, unprotect, unmark, unsusp -->
  <!-- (the state will be set on all specified media) -->
  <!-- mediaID : valid media ID -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The state argument is required exactly once. -->
  <!-- One or more mediaID arguments are required. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="state" value="avail"/>
  <ARGUMENT name="mediaID" value="025311"/>
  <ARGUMENT name="mediaID" value="025312"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetMediaState" statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
 <!-- A status detail element will be returned for each media -->
   <STATUSDETAIL name="mediaID" value="025311"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
   <STATUSDETAIL name="mediaID" value="025312"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetMediaState : public Request
{
public:
   /// Media State values
   enum MediaState
   {
        MEDIA_STATE_TYPE_START=0,
        AVAIL=MEDIA_STATE_TYPE_START,
        UNAVAIL,
        PROTECT,
        UNPROTECT,
        UNMARK,
        UNSUSP,
        MEDIA_STATE_TYPE_END
   };

   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object,
   /// providing it the media name and state.
   SetMediaState(const std::string& inMediaID,
        const MediaState& inState);

   /// Secondary Constructor
   /// This constructor is intended for instantiation of the
   object from a
   /// vector of media ids
   SetMediaState(const MediaList& inMedia,
        const MediaState& inState);

   /// Method to retrieve the map of mediaIDs to their status'
   std::map<std::string, Status> getMediaMap() const;

   /// Method to return the media state
   const MediaState& getMediaState() const;

   /// Method to return the media state as a string
   const std::string& getMediaStateAsString() const;

   /// Method to return the list of media
   MediaList getMediaList() const;
```

```
   /// Method to return the list of local status pairs
   const StatusPairList& getLocalStatus() const;

   /// Method to return the specific local status pair for a given
   mediaID
   const StatusPair& getLocalStatus(const std::string& mediaID)
const;
}
```

SetPolicy

This API allows you to set the number of copies and the maximum number of inactive versions for a policy class.

Input

policyClass: The name of the policy class for which you want to set arguments, or enter “**all**” to apply arguments to all policy classes.

numberOfCopies: The number of copies to maintain for the policy class.

maxInactiveVersions: The maximum of versions to maintain for the policy class.

Note: You must enter one policy class name. You can enter either the number of copies, the maximum number of versions, or both arguments.

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Configure one or all policy classes. -->
<COMMAND name="SetPolicy">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- policyClass : valid policy class name or "all" -->
  <!-- numberOfCopies : 1 .. 8 -->
```

```
  <!-- maxInactiveVersions : 1 .. 10 -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- The policyClass argument is required exactly once. -->
  <!-- At least one of the following arguments are required, but -->
  <!-- no more than one of each: -->
  <!-- numberOfCopies -->
  <!-- maxInactiveVersions -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="policyClass" value="mypolicyclass"/>
  <ARGUMENT name="numberOfCopies" value="2"/>
  <ARGUMENT name="maxInactiveVersions" value="10"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetPolicy" statusCode="0" status="SUCCESS"
        statusDescription="Command Successful">
  <STATUSDETAIL name="policyClass" value="all"
        statusCode="0" status="SUCCESS"
        statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetPolicy : public Request
{
public:
   /// Primary Constructor
   /// This constructor is intended for primary instantiation of
   the object.
   SetPolicy(const std::string& inPolicyClass,
        const uint32_t     inNumberOfCopies,
        const uint32_t     inMaxInactiveVersions);

   /// Secondary Constructor
   /// This constructor is a secondary means to instantiate the
   object.
   SetPolicy(const std::string& inPolicyClass);

   /// Get the policy class name
   const std::string& getPolicyClass() const;

   /// Get the number of copies
```

```
   uint32_t getNumberOfCopies() const;

   /// Get the maximum inactive versions
   uint32_t getMaxInactiveVersions() const;

   /// Return the list of local status pairs.
   const StatusPairList& getLocalStatus() const;

   /// Return a specific local status pair for a given policy
   class
   const StatusPair& getLocalStatus(const std::string&
inPolicyClass) const;
}
```

SetSchedule

This API allows you to specify a schedule for a backup. Running this API affects only system default full backups and default partial backups, not any user-configured backup events.

Input

scheduleType: BACKUP or other scheduled event type.

timeOfDay: The time the scheduled event begins, in 24-hour format (HH:MM).

Output

status: SUCCESS, FAILURE, SUBFAILURE, or SYNTAXERROR status code.

XML Example

Request:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Set the time for the default full backup and default -->
<!-- partial backup schedules. -->
<COMMAND name="SetSchedule">
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- scheduleType : backup -->
  <!-- timeOfDay : time in 24-hour HH:MM format -->
```

```
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <!-- Each argument above is required exactly once. -->
  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
  <ARGUMENT name="scheduleType" value="backup"/>
  <ARGUMENT name="timeOfDay" value="23:59"/>
</COMMAND>
```

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<RESPONSE name="SetSchedule" statusCode="0" status="SUCCESS"
       statusDescription="Command Successful">
  <STATUSDETAIL name="scheduleType" value="backup"
       statusCode="0" status="SUCCESS"
       statusDescription="Command Successful"/>
</RESPONSE>
```

C++ Class Declaration

```
class SetSchedule : public Request
{
public:
   /// Enumerator for the schedule types
   enum ScheduleType {backup=0};

   /// Primary Constructor
   ///                        schedule type.
   SetSchedule(const ScheduleType& inScheduleType,
       const std::string&  inTimeOfDay);

   /// Return the string associated to a ScheduleType
   static std::string getTypeString(const ScheduleType& inType);

   /// Return the schedule type
   const ScheduleType& getScheduleType() const;

   /// Return the scheduled time-of-day
   const std::string& getScheduleTimeOfDay() const;

   /// Return the local status
   const Status& getLocalStatus() const;
}
```

Appendix A
API Example

This appendix contains examples showing how to run a test program for the GetFileAttribute API. Included are two C++ files and two makefiles (one file each for Linux and Solaris platforms). The difference between the two C++ files is that one file is for running the test program via the XML interface. All examples in this appendix apply only to the Storage Manger APIs.

The makefiles are designed to work with the C++ files. Put the C++ source files and the makefile appropriate to your operating system into the same directory and then run the make command to create executable binaries. (The makefiles and C++ files can be found in the directory /usr/adic/SNAPI/examples.)

C++ Test Program Example

Following is a test program for the GetFileAttribute API.

```
//////////////////////////////////////////////////////////////////
// Copyright 2011 Quantum, Inc.
//
// NAME: GetFileAttribute.cc
//
#include <API.hh>

#include <sys/types.h>
#include <unistd.h>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;
using namespace Quantum::SNAPI;
```

```
int
main()
{
  Status status;

  try
  {
    // Create object for requesting file attributes.
    GetFileAttribute getFileAttrReq("/snfs/testFile.dat");

    // Send request to the server and get overall request status code.
    // Note: This method may throw exceptions (see catch block below).
    status = getFileAttrReq.process();

    // Check the returned status code.
    if (status.getCode() != SUCCESS)
    {
      cout << "StatusCode: " << status.getCodeAsString() << endl;
      cout << "Description: " << status.getDescription() << endl;
      cout << "LocalStatus.StatusCode: " <<
            getFileAttrReq.getLocalStatus().getCodeAsString() << endl;
      cout << "LocalStatus.Description: " <<
            getFileAttrReq.getLocalStatus().getDescription() << endl;
    }
    else
    {
      // Get the requested data.
      FileInfo fileInfo = getFileAttrReq.getFileInfo();

      cout << "FileName:        " << fileInfo.getFileName() << endl;
      cout << "Location:        " << fileInfo.getLocationAsString() << endl;
     cout << "Existing Copies: " << fileInfo.getNumberOfExistingCopies() << endl;
      cout << "Target Copies:   " << fileInfo.getNumberOfTargetCopies() << endl;

      MediaList media = fileInfo.getMedia();

      for (int i=0; i<media.size(); i++)
      {
        cout << "Media ID:        " << media[i] << endl;
      }
    }
  }
  catch (SnException& exception)
  {
    switch (exception.getCode())
    {
      case SUBFAILURE:
      case FAILURE:
      case SYNTAXERROR:
      default:
      {
        cout << "Exception code:   " << exception.what() << endl;
        cout << "Exception detail: " << exception.getDetail() << endl;
      }
      break;
```

```
    }
  }
  catch (...)
  {
    cout << "Caught unknown exception." << endl;
  }

  return status.getCode();
}
```

C++ XML Interface Test Program Example

Following is a test program for the GetFileAttribute API run from the XML interface.

```
//////////////////////////////////////////////////////////////
// Copyright 2011 Quantum, Inc.
//
// NAME: GetFileAttributeXML.cc
//

#include <API.hh>

#include <sys/types.h>
#include <unistd.h>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;
using namespace Quantum::SNAPI;

int
main()
{
  Status status;

  // Initialize input command in XML format.
  XML xmlIn("<?xml version=\"1.0\"?>"
            "<COMMAND name=\"GetFileAttribute\">"
            "<ARGUMENT name=\"fileName\" value=\"/snfs/testFile.dat\"/>"
            "</COMMAND>");

  XML xmlOut;

  try
  {
    // Perform the request.
    status = doXML(xmlIn, xmlOut);
  }
  catch (SnException& excptn)
  {
    cerr << "Exception code:   " << excptn.what() << endl;
    cerr << "Exception detail: " << excptn.getDetail() << endl;
```

```
    status = excptn.getCode();
  }

  // Stream the results to standard out.
  cout << xmlOut << endl;

  return status.getCode();
}
```

Makefile Example for Linux Platforms

Following is a makefile example for Linux platforms.

```
#################################################################
# Copyright 2005-2011 Quantum Corporation.
#
# This makefile works on Linux platforms. It looks for libraries and
# headers under /usr/adic/SNAPI/. Use it with the two source files
# listed below.
#
#   GetFileAttribute.cc
#   GetFileAttributeXML.cc
#
# Just rename this file as "makefile" and run "make all".
#
#################################################################
BASEPATH = /usr/adic/SNAPI

# Libraries for linking
LIBSSHARED = -L${BASEPATH}/lib -lsnapi -lpthread
LIBSSTATIC = -L${BASEPATH}/lib/static -lsnapi -lpthread

# Set this to the location of your g++ compiler.
CC = /usr/bin/g++

all:        getFileAttribute.dynamic getFileAttribute.static\
            getFileAttributeXML.dynamic getFileAttributeXML.static

getFileAttribute.dynamic: getFileAttribute.o
                    $(CC) -Wl,-rpath,${BASEPATH}/lib -o $@ getFileAttribute.o
${LIBSSHARED}

getFileAttribute.static: getFileAttribute.o
            $(CC) -o $@ getFileAttribute.o ${LIBSSTATIC}

getFileAttribute.o: GetFileAttribute.cc
            ${CC} -I${BASEPATH}/inc -c $< -o $@

getFileAttributeXML.dynamic: getFileAttributeXML.o
```

```
	$(CC) -Wl,-rpath,${BASEPATH}/lib -o $@ getFileAttributeXML.o
${LIBSSHARED}

getFileAttributeXML.static: getFileAttributeXML.o
	$(CC) -o $@ getFileAttributeXML.o ${LIBSSTATIC}

getFileAttributeXML.o: GetFileAttributeXML.cc
	${CC} -I${BASEPATH}/inc -c $< -o $@

clean:
	/bin/rm -f getFileAttribute.o getFileAttribute.dynamic \
	getFileAttribute.static getFileAttributeXML.o
getFileAttributeXML.dynamic \
	getFileAttributeXML.static
```