REFERENCE MANUAL | PUBLIC
SAP Adaptive Server Enterprise 16.0 SP03
Document Version: 1.0 – 2019-06-06

Reference Manual: Backup Server Archive API

THE BEST RUN SAP

Content

1 Overview

The Backup Server archive API supports a byte stream interface to the back-end of Backup Server. The archive API routines are used by Backup Server to issue I/O requests to an archive byte stream.

The vendor application writer creates the routines documented in this API. These routines exist in a shareable library that is dynamically loaded at run-time. The dump device string of the `DUMP` and `LOAD` commands supports the archive API. This string is used to specify whether the archive device name is a vendor byte stream and provides vendor-specific parameters. The Backup Server boot flag can be used to specify the directory of the sharable libraries. If the boot flag is not specified, the pathname defaults to `$SYBASE/$SYBASE_ASE/lib`.

Related Information

Backup Server Boot Flag [page 18]

1.1 Syntax

The semantics of the string specifying an archive device name provides the byte stream identifier and vendor-specific application information.

The device name can be either a hard-coded literal string or a logical dump device name created with the `sp_addumpdevice` stored procedure. Backup Server detects that the archive device is a byte stream by inspecting the format of the device string. The format of the device string for a byte stream is:

```
streamidentifier::vendor_specific_information
```

Backup Server parses the archive device string and assumes the device is a byte stream if the string begins with `streamidentifier`, followed by a double colon. The archive device string can be up to 127 characters long. The `streamidentifier` name corresponds to the name of the sharable library, which is dynamically loaded at runtime, without the `lib` prefix. The `vendor_specific_information` portion of the device string is only meaningful to a vendor-specific application. Backup Server does not scan this portion of the device string. As such, this vendor specific string can contain any character. It is typically used to uniquely identify a dump.

1.2 Byte Stream Archive Command Options

Many of the options to the `DUMP` and `LOAD` commands are device-dependent.

The valid command options for a byte stream device are: `NOTIFY`, `[INIT | NOINIT]`, and `HEADERONLY` . The `LOAD` command option `with copyonly` is available from versions ASE 15.7 SP137 and ASE 16.0 SP01 PL03.

An error message is issued if any other command options are present for the byte stream device. The `AT` clause is not allowed with a byte stream archive object. If the `AT` clause is specified for a byte stream device, an error is reported and the command aborts. It is assumed that the third-party vendor handles all network services.

The `STRIPE ON` clause is supported, meaning the API supports parallel dump or load through multiple streams opened simultaneously. The vendor application redirects these streams to one or more archive devices. All the streams should be simultaneously available for writing and reading before the dump or load can proceed. Also, the number of stripes specified during a load should match that of the dump. The API writer may choose to disallow multiple stripes by checking the `total_number_of_stripes` field in the `SYB_INFO_T` structure.

Related Information

syb_open [page 10]

2 Application Programming Interface

A pointer to a structure that contains information about the command being executed is passed to the routine used to open the byte stream.

An error or informational text string is passed back from most of the calls when an error condition occurs or when the vendor wants to send an informational message to the client. Part of the information structure that is passed contains the language and character set that is being used by the SAP software. This information is used to localize the messages returned from the routines defined in the API.

Type Definitions

Part of the archive API is a C language include file with definitions used to help the application writer create the routines are part of the archive API. This include file is named `sybackup.h`. The following definitions appear within this include file. All character fields are null terminated strings.

```
#ifndef _sybackup_h_
#define _sybackup_h_
/*
** Copyright Notice and Disclaimer
** -------------------------------
**    (c) Copyright 2016.
**    SAP AG or an SAP affiliate company. All rights reserved.
**    Unpublished rights reserved under U.S. copyright laws.
**
**    SAP grants Licensee a non-exclusive license to use, reproduce,
**    modify, and distribute the sample source code below (the "Sample Code"),
**    subject to the following conditions:
**
**    (i) redistributions must retain the above copyright notice;
**
**    (ii) SAP shall have no obligation to correct errors or deliver
**    updates to the Sample Code or provide any other support for the
**    Sample Code;
**
**    (iii) Licensee may use the Sample Code to develop applications
**    (the "Licensee Applications") and may distribute the Sample Code in
**    whole or in part as part of such Licensee Applications, however in no
**    event shall Licensee distribute the Sample Code on a standalone basis;
**
**    (iv) and subject to the following disclaimer:
**    THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
**    INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
**    AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
**    SAP AG or an SAP affiliate company OR ITS LICENSORS BE LIABLE FOR ANY
**    DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
**    DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
**    SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
**    CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
**    LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
**    OUT OF THE USE OF THE SAMPLE CODE, EVEN IF ADVISED OF THE POSSIBILITY
**    OF SUCH DAMAGE.
*/
#define SYB_MIN_IO          2048
#define SYB_MAX_IO          512 * 1024
```

```
#define SYB_SUCCESS        0
#define SYB_FAILURE        -1
#define SYB_PRE_API_2_1_MAX_STRIPE    32
                        /* maximum number of stripes allowed
                           in pre 2.1 api */
#define SYB_MAX_NAME        30    /* maximum identifier name length */
#define SYB_MAX_STRIPE_NAME    255    /* maximum stripe name length */
#define SYB_API_VERSION_1_0    0x00010000
#define SYB_API_VERSION_1_1    0x00010001
#define SYB_API_VERSION_2_0    0x00020000
#define SYB_API_VERSION_2_1    0x00020001
/* current API version */
#define SYB_API_VERSION        SYB_API_VERSION_2_1
#define SYB_RDONLY    0        /* Read-only mode: LOAD */
#define SYB_WONLY    1        /* Write-only mode: DUMP */
/*
** This structure contains run-time properties of the Backup Server. This is
** used to provide the vendor application with any information about the Backup
** Server that it needs before opening the stripes.
*/
typedef struct
{
    int min_iosize;    /* The smallest I/O size used by the Backup
                  Server. Currently, it is set to 2048 bytes. */
    int max_iosize;    /* The maximum I/O size supported by the Backup
                   Server. */
} SYB_PROP_T;
typedef struct
{
    long    dtdays;        /* number of days since 1/1/1900 */
    long    dttime;        /* number 300th seconds since midnight */
} SYB_API_DATETIME;
typedef struct
{
    long            total_number_of_stripes;
    long            command_type;
/* DATABASE/TRANSACTION/UNKNOWN */
    long            options;
/* bit map of the command/device options. */
    char            database_server_name[SYB_MAX_NAME];
    char            backup_server_name[SYB_MAX_NAME];
    char            database_name[SYB_MAX_NAME];
    char            language[SYB_MAX_NAME];
                    /* language names
                       ** correspond to the subdirectories
                       ** in $SYBASE/locales directory.
*/
    char            character_set[SYB_MAX_NAME];
                    /* character set names
                       ** correspond to the subdirectories
                       ** in $SYBASE/charsets directory.
*/
    SYB_API_DATETIME    timestamp;
    char            **archive_string;
                    /* Array of null terminated archive strings */
    unsigned int    *size;
                    /* An array of estimated number of kilobytes
                      ** per stripe being written to the byte stream.
                       ** This value is only valid when the mode for
                       ** syb_open is set to SYB_WONLY.
*/
} SYB_INFO_T;
/* Command Type definitions for the command_type field in SYB_INFO_T */
#define TRANSACTION    0    /* transaction log dump or load */
#define DATABASE    1        /* database dump or load */
#define CUMULATIVE    2    /* Cumulative dump to dump or load */
#define UNKNOWN        3    /* Pre-API SQL Server in use. Backup Server
                       ** has received a DUMP/LOAD command from a
```

```
                        ** SQL Server which is not passing the
                        ** 'Command Type' to Backup Server. Therefore,
                        ** the Command type can not be determined. */
#define DELTA        4    /* Delta dump to dump or load */
#define INCREMENTAL_DUMP(aux)    (((aux)->at_commandtype == CUMULATIVE) \
               || ((aux)->at_commandtype == DELTA))
/* Option definitions for the options field of SYB_INFO_T */
#define INIT_FLAG       0x0001
/* bit is set if WITH INIT specified on
                        ** the command line, else the bit is not set.
                        ** The vendor may use this flag to indicate
                        ** an existing file may be overwritten or a
                        ** new file created.
*/
#define DEBUG_FLAG      0x0010
/* bit is set if QATRACE_EXT_API is turned on
                        ** in Backup Server. Which indicates that
                        ** user intends to debug external API.
*/
#define SYB_TSM_T_FLAG  0x0100
/* bit is set if TRACE_ENABLE_TSM_MULTITHREAD
                        ** is specified on Backup Server startup. Which
                        ** indicates that user intends TSM
multithread.                    */
/* The remaining bits of this field are reserved for future use. */
#define SYB_MAX_MESSAGE    1024
typedef struct
{
    long    msglen;        /* number of bytes in the message */
    char    message[SYB_MAX_MESSAGE]; /* error/informational message */
} SYB_ERROR_T;
typedef struct
{
        long       api_version;    /* Backup Server Archive API version which the
                           ** vendor application supports. It should
                           ** usually be set to the SYB_API_VERSION
                           ** specified in the include file with which
                           ** the vendor shared library is built.
** However, if the vendor decides to
** support multiple API versions using
** a single shared library, then, they can
** set it to to a version earlier than the
** SYB_API_VERSION specified in the include
** file.
*/
    long        application_version;       /* application version */
    char        name[SYB_MAX_NAME];     /* application name */
    unsigned int     application_options;     /* application options */
    int        app_max_iosize;    /* maximum I/O size supported by
                          ** the vendor application. It
                          ** should be greater than or
                          ** equal to the min_iosize field
                          ** specified in the properties
                          ** parameter of the
                          ** syb_defineapi() call.
*/
} SYB_APPL_T;
/* valid values for application_options */
#define SYB_STRIPE_DESYNC     0x1
/* mode options for syb_close() */
#define SUCCESS_ON_CLOSE    0    /* The stripe is being closed
                           ** successfully
*/
#define ABORT_ON_CLOSE        -1    /* The stripe is being closed
                           ** unsuccessfully or the dump/load
                           ** operation was cancelled.
*/
#endif /* _sybackup_h_ */
```

3 API Routines

The archive API routines are linked into a dynamically loadable library. All routines must be present. If a function is missing, Backup Server reports an error and the command aborts.

Related Information

Dynamically Loadable Library [page 17]

3.1 syb_defineapi

This routine is used to negotiate the capabilities with Backup Server, notably versions and block I/O size. The vendor application writer stores `SYB_API_VERSION` into the `api_version` field of the `SYB_APPL_T` structure. This field provides Backup Server with the current archive API version, which the vendor application is written to. If an unsupported version of the archive API is supplied, Backup Server reports an error and the command aborts.

Syntax

```
int
syb_defineapi(<minversion>, <maxversion>, <error>, <properties>, <args>)
int         minversion;
int        maxversion;
SYB_ERROR_T    *error;
SYB_PROP_T    *properties;
void         *args;
```

Parameters

<minversion>

minimum API version supported.

<maxversion>

maximum API version supported.

<error>

error message text if call fails; informational message if call succeeds.

<properties>

minimum and maximum I/O size used by Backup Server.

<args>

reserved for future use.

Returns

`syb_defineapi()` returns 0 on success; -1 in case of failure.

3.2 syb_queryapi

This routine is used to authenticate the vendor-supplied routines and to get information about the vendor application. The routine retrieves the API version number, vendor name, and application version number. The returned data is used when displaying informational or error messages.
The vendor application writer stores `SYB_API_VERSION` into the `api_version` field of the `SYB_APPL_T` structure. This field provides Backup Server with the current archive API version, which the vendor application is written to. If an unsupported version of the archive API is supplied, then Backup Server reports an error and the command aborts.

Syntax

```
void syb_queryapi(<application_info>)
SYB_APPL_T *appl_info;
```

Parameters

<application_info>

an output parameter. The fields in this vendor information structure are set in this routine.

Returns

`syb_queryapi` always succeeds. `void` is returned.

3.3 set_params

This routine is used to copy the global application arguments with the API.

Syntax

```
int syb_set_params(char **argv);
```

Parameters

argv

parameter is set to the application `argv`. With the exception of the Tivoli Storage Manager (TSM) library, this will always be the `sybmultbuf argv` parameters. The TSM API is also loaded by Backup Server and this variable is used by the API to determine who is loading it.

Returns

`set_params` always succeeds.

3.4 syb_open

This routine allocates the vendor-specific handle and opens the specified byte stream.
The contents of the handle is user-defined.

Syntax

```
void *
syb_open(<stripe_number >, <syb_info>, <mode>, <error>, <args>)
long        stripenum;
SYB_INFO_T    *syb_info;
int        mode;
SYB_ERROR_T    *error;
void         *args;
```

Parameters

<stripe_number>

input parameter that indicates the byte stream to open. The number ranges from `0` to `n-1`, where `n` is the number of stripe used in the `DUMP` or `LOAD` command. This number is an index into the `archive_string` array within the `SYB_INFO_T` structure. The resultant device string corresponds to the byte stream to open.

<syb_info>

input parameter to open.

<mode>

`SYB_RDONLY` if open is for read-only; `SYB_WONLY` for write-only.

<error>

error message text if call fails; informational message if call succeeds.

<args>

reserved for future use.

Returns

`syb_open` returns a pointer to an allocated handle on success. On failure, the routine returns 0 with the error structure set. If an error occurs, the `SYB_ERROR_T` structure is set accordingly. See Error Messages [page 15] for more information.

3.5 syb_close

This routine closes the byte stream pointed to by the `apihandle` parameter.
The routine flushes the byte stream, closes the byte stream, and deallocates the vendor-specific handle.

Syntax

```
int
syb_close(<apihandle>, <mode>, <error>, <args>)
void        *apihandle;
int        mode;
SYB_ERROR_T    *error;
void         *args;
```

Parameters

<apihandle>

api handle to close.

<mode>

mode for closing the byte stream. The two modes are: `SUCCESS_ON_CLOSE`; `ABORT_ON_CLOSE`. If `SUCCESS_ON_CLOSE` is specfied, the stream can be closed under the assumption that the dump or load operation succeeded. If `ABORT_ON_CLOSE` is specified, an error or interruption was encountered within the dump or load operation. The API application can take the required actions. The default value is `SUCCESS_ON_CLOSE`.

<error>

error message text if call fails; informational message if call succeeds.

<args>

reserved for future use.

Returns

`syb_close` can return the following values:

Returns	Indicates
SYB_SUCCESS	Completed successfully.
SYB_FAILURE	Failed and error structure set.

If an error occurs, the `SYB_ERROR_T` structure is set accordingly. See Error Messages [page 15] for more information.

3.6 syb_read

This routine reads from the byte stream specified by the `handle` parameter.
This routine reads the size number of bytes from the byte stream and places these bytes into the memory addressed by the buffer.

Syntax

```
int
syb_read(<apihandle>, <buffer>, <size>, <error>, <args>)
```

```
void        *apihandle;
char        *buf;
long        size;
SYB_ERROR_T    *error;
void         *args;
```

Parameters

<apihandle>

api handle returned from `syb_open` call.

<buffer>

pointer to buffer where data is read into.

<size>

size number of bytes to read. The size ranges from `SYB_MIN_IO` to `SYB_MAX_IO`.

<error>

error message text if call fails; informational message if call succeeds.

<args>

reserved for future use.

Returns

`syb_read` can return the following values:

Returns	Indicates
SYB_SUCCESS	Completed successfully.
SYB_FAILURE	Failed and error structure set.

If a failure occurs and the size number of bytes cannot be read, then zero bytes are read from the byte stream. For example, the contents of the buffer are ignored and the `SYB_ERROR_T` structure is set accordingly. See Error Messages [page 15] for more information.

3.7 syb_write

This routine writes to the byte stream pointed to by the handle parameter.
This routine writes the number of bytes, which is defined using `size`, from the memory pointed to by `buffer` to the byte stream. If a failure occurs, for example, if the number of bytes (using `size`) is not written, then zero bytes are written to the byte stream and the`SYB_ERROR_T` structure is set accordingly. See Error Messages [page 15] for more information.

Syntax

```
int
syb_write(<apihandle>, <buffer>, <size>, <error>, <args>)
void        *apihandle;
char        *buf;
long        size;
SYB_ERROR_T    *error;
void        *args;
```

Parameters

<apihandle>

API handle returned from `syb_open` call.

<buffer>

pointer to the buffer to which you want to write.

<size>

size of the data to write. The size ranges from `SYB_MIN_IO` to `SYB_MAX_IO`.

<error>

error message text if call fails; informational message if call succeeds.

<args>

reserved for future use.

Returns

`syb_write` can return the following values:

Returns	Indicates
SYB_SUCCESS	Completed successfully.
SYB_FAILURE	Failed and error structure set.

4 Error Messages

An error or informational message can be displayed to the client or system console when an API call returns. If the `msglen` field of the `SYB_ERROR_T` structure is greater than zero, then the contents of the message field is displayed.

The type of message is determined by the return status of the API call. If `SYB_FAILURE` is returned, then message contains an error message. If `SYB_SUCCESS` is returned, then message contains an informational message. For both error and informational messages, the `msglen` field of `SYB_ERROR_T` must contain the number of bytes in the message. The displayed output contains the application name, application version (which was returned from the `syb_queryapi()` routine), name of the routine that failed, and the message returned from the API call.

4.1 Localization

All locales information (language and character set name) that is passed to `syb_open()` uses the server client's locale.

If an error message is returned from one of the API functions, and this error message is reported to some destination other than the client, then this alternate destination receives these messages in the locales of the client. This is a restriction of the current version of the Backup Server archive API. Errors that the `sybmultbuf` process reports to the Backup Server error log are not localized. All of these message are reported in US English and the native character set of the platform. This restriction is imposed by the architecture of Backup Server. For a list of `sybmultbuf` messages, see Error and Informational Messages [page 15].

4.2 Error and Informational Messages

During the initialization phase of the archive API, errors detected as part of the integration of the archive API have their error messages reported to the Backup Server error log.

This refers to errors that occur when the `sybmultbuf` process starts, and includes errors that occur due to accessibility problems with the API library, problems with loading the API functions, or errors returned from `syb_queryapi()`.

Backup Server Error and Informational Messages

The following error or information messages are returned to the client when dumping or loading data pages to the archive API.

- Archive API error for device=devicename: Application=applicationname, Library version=libraryver- sion, API routine=routinename, Message=errormessage.
- The diagnostic option specifier must be an integer.
- Archive API information for device=devicename: Application=applicationname, Library ver- sion=libraryversion, Message=errormessage.
- Remote Backup Server access is not allowed when interfacing to a stripe for device 'devicename'.
- Unable to get current date and time. Internal error.
- Unable to open API library for device 'devicename'. Library path is 'librarypathname'. Error code=errorcode.
- Unable to load API function functionname for device 'devicename'. Code=errorcode, Message=message.
- Archive API version versionnumber is not supported for device 'devicename'."

Sybmultbuf Messages

The following error or information messages are reported to the Backup Server error log when dumping or loading data pages to the archive API.

- Cannot allocate memory for resource 'resourcename'.
- Failed to open database info file filename. Code=errorcode, Message=message.
- Failed to read database info file filename. Code=errorcode, Message=message.
- Total number of stripes does not match the number of archive strings in the database info file. Total stripes=number, number archive strings=number.
- Cannot open Backup Server error log 'errorlogname'. Code=errorcode, Message=message.
- Cannot write to Backup Server error log 'errorlogname'. Code=errorcode, Message=message.
- Cannot seek to end of Backup Server error log 'errorlogname'. Code=errorcode, Message=message.
- Emulator Diagnostics: diagnosticmessage.

5 Dynamically Loadable Library

All of the API routines are linked together to create a dynamically loadable library.

The naming convention for the dynamically loadable library is:

`lib <vendoridentifier>.<libsuffix>`

where `<vendoridentifier>` is the stream identifier portion of the archive device string, located to the left of the double colon. The `<libsuffix>` portion is the shared library suffix appropriate for the platform which Backup Server is running on.

The following table lists each suffix per platform, plus the compilation and linker command arguments needed to create a sharable library:

Platform	Shared Library Suffix	Compile Flags	Link Flags
HPIA	`.so`	`-c +DD64`	`-b -E`
AIX	`.so`	`-c -q64`	`-bM:SRE -bnoentry -G -b64 -bexpall -ldl -lm -lc`
Solaris	`.so`	`-c -m64 -xcode=pic13 -Xa`	`-dy -G`
Sunx64	`.so`	`-c -m64 -fPIC -Xa`	`-dy -G`
Linux	`.so`	`-c -m64 -fPIC`	`-G`
IBM plinux	`.so`	`-c -q64 -qpic=small`	`-G -q64`
Windows	`.DLL`	`/MD` (Microsoft Visual C++)	`/DLL` (Microsoft Visual C++)

If this flag is not specified, the directory defaults to `$SYBASE/lib`.

On the Windows platform, define the API functions to use the standard calling convention, in which the called function cleans up the stack before returning. For example, use the prefix `stdcall` when compiling with the Microsoft Visual C++ compiler. Refer to your compiler documentation for an equivalent option. Your linker may also need an export file containing the API functions. For example, the Visual C++ linker requires the `/DEF:` option, which specifies the `.DEF` file that contains the export definition for all the API functions, as shown in this DEF file for `libcompress.dll`:

```
LIBRARY compress
EXPORTS
        syb_queryapi    @1
        syb_open        @2
        syb_read        @3
        syb_write       @4
        syb_close       @5
        syb_defineapi   @6
```

Refer to your linker documentation for more details.

6 Backup Server Boot Flag

A flag is available to specify the directory where the byte stream sharable libraries resides.

This flag is defined as:

```
-A<directory path>
```

If this flag is not specified, the directory defaults to `$SYBASE/lib`.

7 System Stored Procedures

When using the `sp_addumpdevice` system stored procedure to access a byte stream device, use `disk` and not `tape`. The `size` parameter is ignored for byte stream devices.

8 Diagnostics

This section describes the diagnostic output that is printed to the error log and the tracing messages that are reported prior to calling an API function and again after returning from the API function.

8.1 Command Line Option

To instruct Backup Server to print API diagnostics, use the `-D1` command line option.

Before and after each API function, a trace message is written to the Backup Server error log. It is useful for determining if a `DUMP` or `LOAD` command is hung within an API call, or whenever an abnormal condition occurs within an API function. Besides the diagnostics information, the API will set a `DEBUG_FLAG 0x0010` set in the structure `SYB_INFO_T`, in the field `syb_info.syb_info_2_1.options` that is passed to all the API functions. This allows API developers to add their own tracing information.

8.2 Diagnostic Output

The `sybmultbuf` subprocesses prints all diagnostic messages to the Backup Server error log.

When the appropriate diagnostics flag is specified, a message is written to the Backup Server error log. The message has the following format:

`<datetime>:<stripeid><P>: <message_text>`

Where:

`<datetime>`	is the date and time of when the message is being written to the error log. The format is `<MMM> <DD>< hh:mm:ss> <YYYY>`, where `<MMM>` is the abbreviated month name, `<DD>` is the day of month,`< hh>` is the hour (military), `<mm>` is minutes, `<ss>` is seconds, and `<YYYY>` is the year. For example: Oct 12 11:53:48 1994
`<stripeid>`	is a two-digit number for the ordinal stripe id of the `DUMP` and `LOAD` commands. This helps to identify the originator of each message.
`<P>`	specifies which of the `sybmultbuf` processes issued the message. 'A' indicates the archive `sybmultbuf` and 'D' indicates the database `sybmultbuf`.
`<message_text>`	is the diagnostic message text.

8.2.1 Tracing Information

Whenever the diagnostic option is specified, tracing messages are reported prior to calling an API function and again after returning from the API function. The precise text for each tracing message differs for each API function. The information reported within each tracing message contains parameter information passed to the API function or results returned from the API function.

API Function	Prior to Call	After Return
`syb_open`	• Function being called • Stripe number • Open mode – READ or WRITE • Total number of stripes • Command type – DATABASE, CUMULATIVE, TRANSACTION, or UNKNOWN • Options • Database server name • Backup Server name • Database name – name of the database being dumped or loaded • Language name – the client's language name • Character set name – the client's character set name • Timestam p – in the format `<YYY MMD hhmmsslll>`, where `<YYYY>` if year, `<MM>` is month, `<DD>` is day, `<hh>` is hour (in 24 hour format),`< mm>` is minutes, `<ss>` is seconds, and `<lll>` is milliseconds. • Archive device strings – each archive device string is displayed	• Function returning • Return status • Error message – reported only if return status is `SYB_FAIL` • API handle – hexadecimal value of the returned handle
`syb_close`	• Function being called • API handle	• Function returning • return status • Error message – reported only if return status is `SYB_FAIL`
`syb_read`	• Function being called • API handle • Number of bytes to read	• Function returning • Return status • Error message – reported only if return status is `SYB_FAIL`
`syb_write`	• Function being called • API handle • Number of bytes to write	• Function returning • Return status • Error message – reported only if return status is SYB_FAIL

API Function	Prior to Call	After Return
`syb_queryapi`	• Function being called	• Function returning • Return status • API version • Application version • Application name

9 Programming Example

In this example, the routines read and write to a UNIX named pipe.

This example shows the C code that implements the functions defined by the API:

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/param.h>
#include <sybackup.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define LIB_VERSION 2
#define APPL_NAME "Pipe example"
int
syb_defineapi(minversion, maxversion, error, properties)
int         minversion;
int        maxversion;
SYB_ERROR_T    *error;
SYB_PROP_T    *properties;
{
        return SYB_SUCCESS;
}
void
syb_queryapi(SYB_APPL_T *app_info)
{
    /* Set the api version number */
    app_info->api_version = SYB_API_VERSION;
    /* version of this code */
    app_info->application_version = LIB_VERSION;
    /* version of this code */
    app_info->app_max_iosize = 65536;
    /* copy this application name and null terminate */
    strncpy(app_info->name, APPL_NAME, sizeof(app_info->name));
}
void *
syb_open(int stripe_number, SYB_INFO_T *syb_info, int mode, SYB_ERROR_T *error)
{
    int    pipe_fd;
    int    *handle;
    char    *pipe_name;
    /*
    ** the format for the archive string is
    ** pipedev::pipe_path_name. The syb_info
    ** archive_string[stripe_number] contains
    ** this string that was passed in from the
    ** command line. This routine does no error
    ** checking of the format of the string and
    ** simply attempt to open the pipe.
    */
    if ((pipe_name = strstr(syb_info->archive_string[stripe_number], "::"))
        != (char *)NULL)
    {
        pipe_name += 2;
        if (strlen(pipe_name) == 0)
        {
            snprintf(error->message,sizeof(error->message),
                "Invalid device name %s.",
                syb_info->archive_string[stripe_number]);
                error->msglen = strlen(error->message);
```

```
            return(0);
        }
    }
    if (mode == SYB_RDONLY)
    {
        pipe_fd = open(pipe_name, O_RDONLY, 0);
    }
    else
    {
        pipe_fd = open(pipe_name, O_WRONLY, 0);
}
    if (pipe_fd < 0)
    {
        snprintf(error->message,sizeof(error->message),
            "open() error errno = %d device %s", errno, pipe_name);
        error->msglen = strlen(error->message);
        return(0);
    }
    if (pipe_fd < 0)
    {
        snprintf(error->message,sizeof(error->message),
            "open() error errno = %d device %s", errno, pipe_name);
        error->msglen = strlen(error->message);
        return(0);
    }
    /*
    ** Allocate memory to save the file handle
    ** that is passed to the syb_read, syb_write,
    ** and syb_close routines.
    */
    handle = (int *) malloc(sizeof(pipe_fd));
    if (handle == 0)
    {
        snprintf(error->message,sizeof(error->message),
            "Unable to allocate memory (malloc)");
        error->msglen = strlen(error->message); return(0);
    }
    *handle = pipe_fd;
    return((void *) handle);
}
int
syb_read(int *handle, char *buffer, long size, SYB_ERROR_T*error)
{
    int    num_read;
    int    total_read;
    /*
    ** This routine reads the number of bytes
    ** requested. It may make several read requests
    ** if the read call returns less than the number
    ** of bytes requested.
    */
    total_read = 0;
    while ( (size - total_read)> 0)
    {
        num_read = read(*handle,&buffer[total_read], size-total_read);
        if (num_read < 0)
        {
            snprintf(error->message,sizeof(error->message),
                "read() from pipe failed errno=%d", errno);
            error->msglen = strlen(error->message);
            return(SYB_FAILURE);
        }
        if (num_read == 0)
        {
            snprintf(error->message,sizeof(error->message),
                "read from pipe failed no writer");
            error->msglen = strlen(error->message);
            return(SYB_FAILURE);
```

```
        }
        total_read = total_read + num_read;
    }
    return(SYB_SUCCESS);
}
int
syb_write(int *handle, char *buffer, long size, SYB_ERROR_T*error)
{
    int    num_written;
    num_written = write(*handle, buffer, size);
    if (num_written != size)
    {
        if (num_written < 0)
        {
            snprintf(error->message, sizeof(error->message),
                "write() error errno=%d",errno);
        }
        else
        {
            sprintf(error->message,
                "Attempt to write %d bytes failed; "
                "written only %d", size, num_written);

            error->msglen = strlen(error->message);
            return(SYB_FAILURE);
        }
    }
    return(SYB_SUCCESS);
}
int
syb_close(int *handle, int mode, SYB_ERROR_T *error)
{
    int    retcode;
    int    offset;
    retcode = close(*handle);
    if (retcode < 0)
    {
        snprintf(error->message, sizeof(error->message),
            "close() error errno=%d",errno);
    }
    else /* Init buffer for subsequent error messages */
    {
        error->message[0] = '\0';
        error->msglen = strlen(error->message);
    }
    /* Free memory that was allocated for the  handle.*/
    free(handle);
    return (retcode < 0) ? SYB_FAILURE : SYB_SUCCESS;
}
```

9.1 Build and Library Location

Compile and link the C code into a sharable library. The library name is important, since this is also the identifier used in the archive device name string to invoke the routines in this shareable library.

Move the library to either the default path`$SYBASE/lib`, or to a user-specified location. In the provided programming example, the name of the shared library is `libpipedev.so`.

10 Dump Command

Use the `DUMP` command to invoke the I/O routines in the `libpipedev.so` library and access to the named pipe `/tmp/pipedevice`.

The command is:

```
DUMP DATABASE foo TO “pipedev::/tmp/pipedevice”
```

The `sp_addumpdevice` system stored procedure defines a dump device in the server:

```
sp_addumpdevice “disk”,dumpdev1, “pipedev::/tmp/pipedevice”
DUMP DATABASE foo TO dumpdev1
```

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon  : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon  : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

www.sap.com/contactsap

THE BEST RUN SAP