SYBASE®

Common Libraries Reference Manual

Open Client™ and Open Server™

15.5

DOCUMENT ID: DC32850-01-1550-01

LAST REVISED: October 2009

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

This book contains reference information regarding:

- The C version of CS-Library, which contains utility routines that are useful to both Open Client™ Client-Library™ and Open Server™ Server-Library applications.
- The C version of Bulk-Library, which provides bulk copy routines for Client-Library and Server-Library applications. Bulk copy allows high-speed transfer of data between a database table and program variables.

Note In previous Open Client and Open Server releases, Bulk-Library was referred to as "the Bulk Copy routines."

Audience

This manual is designed to serve as a reference manual for programmers who are writing Client-Library or Open Server applications. It is written for application programmers who are familiar with the C programming language.

How to use this book

This book contains these chapters:

- Chapter 1, "Introducing CS-Library," contains a brief introduction to CS-Library.
- Chapter 2, "CS-Library Routines," contains specific information about each CS-Library routine, such as what parameters the routine takes and what it returns.
- Chapter 3, "Bulk-Library," contains a brief introduction to Bulk-Library.
- Chapter 4, "Bulk-Library Routines," contains specific information on each Bulk-Library routine.

Related documents

You can see these books for more information:

- The *Open Server Release Bulletin for Microsoft Windows* contains important last-minute information about Open Server.

- The *Software Developer's Kit Release Bulletin for Microsoft Windows* contains important last-minute information about Open Client and SDK.
- The *jConnect for JDBC Release Bulletin* versions 6.05 and 7.0 contains important last-minute information about jConnect™.
- The *Open Client and Open Server Configuration Guide for Microsoft Windows* contains information about configuring your system to run Open Client and Open Server.
- The *Open Client Client-Library/C Reference Manual* contains reference information for Open Client Client-Library.
- The *Open Client Client-Library/C Programmers Guide* contains information on how to design and implement Client-Library applications.
- The *Open Server Server-Library/C Reference Manual* contains reference information for Open Server Server-Library.
- The *Open Client and Open Server Programmers Supplement for Microsoft Windows* contains platform-specific information for programmers using Open Client and Open Server. This document includes information about:
  - Compiling and linking an application
  - The sample programs that are included with Open Client and Open Server
  - Routines that have platform-specific behaviors
- The *jConnect for JDBC Installation Guide* version 6.05 contains installation instructions for jConnect for JDBC™.
- The *jConnect for JDBC Programmers Reference* describes the jConnect for JDBC product and explains how to access data stored in relational database management systems.
- The *Adaptive Server Enterprise ADO.NET Data Provider Users Guide* provides information on how to access data in Adaptive Server® using any language supported by .NET, such as C#, Visual Basic .NET, C++ with managed extension, and J#.
- The *Adaptive Server Enterprise ODBC Driver by Sybase Users Guide* for Windows and Linux, provides information on how to access data from Adaptive Server on Microsoft Windows, Linux, and Apple Mac OS X platforms, using the Open Database Connectivity (ODBC) Driver.

- The *Adaptive Server Enterprise OLE DB Provider by Sybase Users Guide for Microsoft Windows* provides information on how to access data from Adaptive Server on Microsoft Windows platforms, using the Adaptive Server OLE DB Provider.

Other sources of information

Use the Sybase® Getting Started CD, the SyBooks™ CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click Partner Certification Report.
3. In the Partner Certification Report filter select a product, platform, and timeframe and then click Go.

4 Click a Partner Certification Report title to display the report.

❖ **Finding the latest information on component certifications**

1 Point your Web browser to Availability and Certification Reports at http://certification.sybase.com/.

2 Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.

3 Select Search to display the availability and certification report for the selection.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1 Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.

2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.

3 Select a product.

4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.

5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

Table 1: Syntax conventions

Key	Definition
command	Command names, command option names, utility names, utility flags, and other keywords are in sans serif font.
variable	Variables, or words that stand for values that you fill in, are in *italics*.
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include the braces in the command.
[ ]	Brackets mean choosing one or more of the enclosed items is optional. Do not include the braces in the command.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you can select only one of the options shown.
,	The comma means you can choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Open Client and Open Server documentation has been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Introducing CS-Library

This chapter gives an overview of CS-Library. It covers the following topics:

CS-Library overview

CS-Library provides utility routines for use in application program development to support:

- Datatype conversion
- Arithmetic operations
- Character-set conversion
- Datetime operations
- Sort-order operations
- Localized error messages

CS-Library also includes routines to allocate and deallocate CS-Library structures.

Although you can write a standalone CS-Library application, CS-Library's primary function is to provide common utility routines to Client-Library and Server-Library applications.

Because Client-Library and Server-Library programs require a context structure, which can only be allocated using CS-Library, all Client-Library and Server-Library programs include at least two calls to CS-Library—one to allocate a CS_CONTEXT and one to deallocate it.

A context structure contains information about an application's runtime environment, or "context." See "Structures" on page 3.

Using CS-Library

You can call CS-Library routines either from within a Client-Library or Server-Library application, or from within a standalone CS-Library application.

Open Client and Open Server applications

Typically, CS-Library routines are called from within a Client-Library or Server-Library application.

Because the Client-Library and Server-Library header files *ctpublic.h* and *ospublic.h* include the CS-Library header file *cspublic.h*, Client-Library and Server-Library applications do not need an additional header file to make CS-Library calls.

After calling cs_ctx_alloc to allocate a CS_CONTEXT, a Client-Library or Server-Library application is free to call any other CS-Library routine.

A standalone CS-Library application

It is possible to write a standalone CS-Library application, although this is not a typical use of CS-Library. For example, a standalone application might make CS-Library calls to use the Open Client and Open Server datatypes and datatype conversion routines.

This type of application needs to include the standard CS-Library header file, *cspublic.h*.

The *Open Client and Open Server Programmers Supplement* includes compiling and linking instructions for CS-Library on your platform.

Structures

CS-Library makes use of several structures, including the CS_CONTEXT control structure, the CS_DATAFMT data format structure, and the CS_LOCALE locale information structure.

The CS_CONTEXT structure is a hidden structure whose internals are not available to an application. The CS_CONTEXT is discussed briefly in the following section.

The CS_CONTEXT structure is also required for Client-Library and Server-Library applications.

- For more information about how Client-Library uses the CS_CONTEXT structure, see the *Open Client Client-Library/C Reference Manual* or the *Open Client Client-Library/C Programmers Guide*.
- For more information about how Server-Library uses the CS_CONTEXT structure, see the *Open Server Server-Library/C Reference Manual*.

The CS_DATAFMT and CS_LOCALE structures are documented in Chapter 2, “Topics,” in the *Open Client Client-Library/C Reference Manual*.

CS_CONTEXT structure

CS-Library defines a single control structure, CS_CONTEXT.

A CS_CONTEXT structure stores configuration information that describes a particular programming context. An application must allocate a CS_CONTEXT structure before calling any other Client-Library, Server-Library, or CS-Library routine.

An application allocates a CS_CONTEXT structure by calling cs_ctx_alloc or cs_ctx_global.

An application can customize a CS_CONTEXT by changing the values of context properties. The following routines change the values of context properties:

- The CS-Library routine cs_config (after the context has been allocated)
- The Client-Library routine ct_config (after the Client-Library routine ct_init has been called for the context)
- The Server-Library routine srv_props (after calling the Server-Library routine srv_version for the context)

An application should deallocate all existing context structures before exiting. An application deallocates a CS_CONTEXT structure by calling cs_ctx_drop.

Datatypes, constants, and conventions

CS-Library uses the same datatypes, constants, and conventions as Client-Library and Server-Library and can be found in the following documents:

- The “Using Open Client and Open Server Datatypes” chapter in the *Open Client Client-Library/C Programmers Guide*
- The “Types” section in the *Open Client Client-Library/C Reference Manual*
- The “Types” section in the *Open Server Server-Library/C Reference Manual*

Error handling

All CS-Library routines return success or failure indications. Sybase strongly recommends that applications check these return codes.

In addition, CS-Library routines can generate CS-Library messages, which range in severity from informational messages to fatal errors. Applications should take steps to receive and handle these messages. In most cases, when a CS-Library routine fails, CS-Library generates a message that describes the reason for the failure.

Two methods of handling messages

An application can handle CS-Library messages in one of two ways:

- By installing a callback routine to handle messages
- Inline, using the CS-Library routine cs_diag

The callback method has the following advantages:

- Gracefully handles unexpected errors

CS-Library automatically calls the appropriate message callback routine whenever a message is generated, so an application can trap unexpected errors. An application using only inline error-handling logic may not successfully trap errors that have not been anticipated.

- Centralizes message-handling code

  Since all errors are handled in the callback, there is no need to add inline message-handling code after each CS-Library call.

Inline message handling has the advantage of allowing an application to check for messages at particular times. For example, an application that makes a sequence of calls to establish a connection might wait until the connection-related call sequence is complete before checking for messages.

Most applications use the callback method to handle messages.

An application indicates which method it will use for a particular context either by calling cs_config to install a message callback routine or by calling cs_diag to initialize inline message handling.

An application can switch back and forth between the inline method and the callback method:

- Installing a message callback routine turns off inline message handling. Any saved messages are discarded.
- Likewise, calling cs_diag to initialize inline message handling "de-installs" the application's CS-Library message callback. As a result, the application's first CS_GET call to cs_diag will retrieve a warning message to this effect.

If a message callback is not installed and inline message handling is not enabled, CS-Library discards message information.

Using a callback to handle messages

To handle CS-Library errors with a callback function, your application must:

- Declare the callback function as described in "Defining a CS-Library message callback" on page 6.
- Install the callback error handler by calling cs_config to set the CS_MESSAGE_CB property. For a detailed description, see "CS-Library Message Callback property" on page 21.

Defining a CS-Library message callback

A CS-Library message callback is defined as follows:

```
CS_RETCODE CS_PUBLIC cslibmsg_cb(context, message)
CS_CONTEXT      *context;
CS_CLIENTMSG   *message;
```

where:

- *context* is a pointer to the CS_CONTEXT structure for which the message occurred.
- *message* is a pointer to a CS_CLIENTMSG structure containing message information. For information on the CS_CLIENTMSG structure, see the "CS_CLIENTMSG Structure" topics page in the *Open Client Client-Library/C Reference Manual*. Note the following similarities with Client-Library:
  - Error severities for CS-Library errors have the same meaning as for Client-Library errors.
  - The *message->msgnumber* field is a bit-packed CS_INT. This number is unpacked with the macros CS_LAYER, CS_ORIGIN, CS_NUMBER, and CS_SEVERITY. This method is the same for Client-Library messages.

  Note that *message* can have a new value each time the message callback is called.

A CS-Library message callback must return either:

- CS_SUCCEED, to instruct CS-Library to continue any processing that is currently occurring on this context, or
- CS_FAIL, to instruct CS-Library to terminate any processing that is currently occurring on this context.

CS-Library message callback example

```
/*
 **  cslib_err_handler() - CS-Library error handler.
 **
 **   This routine is the CS-Library error handler used by this
 **   application. It is called by CS-Library whenever an error
 **   occurs. Here, we simply print the error and return.
 **
 **  Parameters:
 **    context
```

```
**      A pointer to the context handle for context
**      on which the error occurred.
**    error_msg
**      The structure containing information about the
**      error.
**
**  Returns:
**      CS_SUCCEED
*/
CS_RETCODE CS_PUBLIC cslib_err_handler(context, errmsg)
CS_CONTEXT    *context;
CS_CLIENTMSG  *errmsg;
{
  /*
  ** Print the error details.
  */
  fprintf(stdout, "CS-Library error: ");
  fprintf(stdout, "LAYER = (%ld) ORIGIN = (%ld) ",
          CS_LAYER(errmsg->msgnumber),
          CS_ORIGIN(errmsg->msgnumber) );
  fprintf(stdout, "SEVERITY = (%ld) NUMBER = (%ld)\n",
          CS_SEVERITY(errmsg->msgnumber),
          CS_NUMBER(errmsg->msgnumber) );
  fprintf(stdout, "\t%s\n", errmsg->msgstring);
  /*
  ** Print any operating system error information.
  */
  if( errmsg->osstringlen > 0 )
  {
    fprintf(stdout, "CS-Library OS error %ld - %s.\n",
            errmsg->osnumber, errmsg->osstring);
  }
   /*
  ** All done.
  */
  return (CS_SUCCEED);
}
```

Inline message handling

An application calls cs_diag to initialize inline CS-Library message handling for a context.

An application that is retrieving messages into SQLCA, SQLCODE, or SQLSTATE must set the CS-Library property CS_EXTRA_INF to CS_TRUE.

For information on the inline method of handling CS-Library messages, see the reference page for cs_diag in Chapter 2, “CS-Library Routines.”

CHAPTER 2

CS-Library Routines

This chapter contains a reference page for each CS-Library routine.

Routines	Description	Page
cs_time	Retrieves the current time.	93
cs_validate_cb	A Client-Library callback routine, registered through ct_callback.	95
cs_will_convert	Indicates whether a specific datatype conversion is available in the Client/Server libraries.	96

cs_calc

Description

Performs an arithmetic operation on two operands.

Syntax

```
CS_RETCODE cs_calc(context, op, datatype, var1,
        var2, dest)

CS_CONTEXT      *context;
CS_INT          op;
CS_INT          datatype;
CS_VOID         *var1;
CS_VOID         *var2;
CS_VOID         *dest;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

op

One of the following symbolic values:

Value of op	Arithmetic operation	*dest Value on return
CS_ADD	Addition	var1 + var2
CS_SUB	Subtraction	var1 - var2
CS_MULT	Multiplication	var1 * var2
CS_DIV	Division	var1 /var2

datatype
One of the following symbolic values, to indicate the datatype of var1, var2, and dest:

Value of datatype	Indicates this datatype
CS_DECIMAL_TYPE	CS_DECIMAL
CS_MONEY_TYPE	CS_MONEY
CS_MONEY4_TYPE	CS_MONEY4
CS_NUMERIC_TYPE	CS_NUMERIC

*var1, *var2, and *dest must all be the same datatype as indicated by the value of *datatype*.

var1
A pointer to the first operand for the arithmetic operation.

var2
A pointer to the second operand for the arithmetic operation.

dest
A pointer to a destination buffer. If cs_calc returns CS_FAIL, **dest* is not modified.

Return value

cs_calc can return the following values:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Common reasons for a cs_calc failure include:

- An invalid parameter
- Attempted division by 0
- Destination overflow

cs_calc generates a CS-Library error message for most failure conditions. See "Error handling" on page 4.

Usage

- *var1*, *var2*, and *dest* must have the same datatype, as indicated by the datatype parameter.
- In case of error, **dest* is not modified.

See also

cs_convert

cs_cmp

Description

Compares two data values.

Syntax

```
CS_RETCODE cs_cmp(context, datatype, var1, var2,
                  result)
CS_CONTEXT      *context;
CS_INT          datatype;
CS_VOID          *var1;
CS_VOID          *var2;
CS_INT          *result;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

datatype

One of following symbolic values, to indicate the datatype of var1 and var2:

Value of datatype	Indicates this datatype
CS_DATE_TYPE	CS_DATE
CS_TIME_TYPE	CS_TIME
CS_DATETIME_TYPE	CS_DATETIME
CS_DATETIME4_TYPE	CS_DATETIME4
CS_DECIMAL_TYPE	CS_DECIMAL
CS_MONEY_TYPE	CS_MONEY
CS_MONEY4_TYPE	CS_MONEY4
CS_NUMERIC_TYPE	CS_NUMERIC
CS_BIGDATETIME_TYPE	CS_BIGDATETIME
CS_BIGTIME_TYPE	CS_BIGTIME

var1

A pointer to the first operand for the comparison.

var2

A pointer to the second operand for the comparison.

result

On successful return, **result* is set to indicate the result of the comparison:

Value of *result	Indicates
-1	*var1* is less than *var2*.
0	*var1* is equal to *var2*.
1	*var1* is greater than *var2*.

Return value

cs_cmp can return the following values:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. If cs_cmp returns CS_FAIL, **result* is undefined.

The most common reason for a cs_cmp failure is an invalid parameter.

cs_cmp generates a CS-Library error message for most failure conditions. See "Error handling" on page 4.

Usage

- cs_cmp sets **result* to indicate the result of the comparison.
- *var1* and *var2* must have the same datatype, as indicated by the *datatype* parameter.
- To compare string values, an application can call cs_strcmp.

See also

cs_calc, cs_convert, cs_strcmp

cs_config

Description

Set or retrieve CS-Library properties.

Syntax

```
CS_RETCODE cs_config(context, action, property,
                     buffer, buflen, outlen)

CS_CONTEXT      *context;
CS_INT          action;
CS_INT          property;
CS_VOID         *buffer;
CS_INT          buflen;
CS_INT          *outlen;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

action

One of the following symbolic values:

action	cs_config
CS_SET	Sets the value of the property.
CS_GET	Retrieves the value of the property.
CS_CLEAR	Clears the value of the property by resetting it to its default value.

property
The property whose value is being set or retrieved, according to the following table:

Table 2-1: Values for cs_config property parameter

Value of property	Controls	Action	*buffer is
CS_APPNAME	The name the application calls itself.	Set, retrieve, or clear.	A CS_CHAR string. The default is NULL.
CS_CONFIG_FILE	The name and path of the Open Client and Open Server runtime configuration file. Meaningful only when external configuration has been enabled by setting CS_EXTERNAL_CONFIG.	Set, retrieve, or clear.	A CS_CHAR string. The default is NULL, which means a platform-specific default is used. See "Runtime configuration file property" on page 19.
CS_DEFAULT_IFILE	The name and path to an alternate default interfaces file.	Set, retrieve, or clear.	A CS_CHAR string to the new interfaces file.
CS_EXTERNAL_CONFIG	Whether or not the Client-Library routine ct_init reads an external configuration file to set default property values.	Set, retrieve, or clear.	CS_TRUE or CS_FALSE. The default, CS_TRUE, is dependent on whether the external configuration file exists. See "External configuration property" on page 20.
CS_EXTRA_INF	Whether or not to return the extra information that is required when processing messages inline using a SQLCA, SQLCODE, or SQLSTATE structure.	Set, retrieve, or clear.	CS_TRUE or CS_FALSE. CS_FALSE is the default.

Value of property	Controls	Action	*buffer is
CS_LIBTCL_CFG	The name and path to an alternate *libtcl.cfg* file.	Set, retrieve, or clear.	A CS_CHAR string to the new *libtcl.cfg* configuration file.
CS_LOC_PROP	A CS_LOCALE structure that defines localization information for this context.	Set, retrieve, or clear.	A CS_LOCALE structure previously allocated by the application.
CS_MESSAGE_CB	The CS-Library message callback routine, which is an application-provided handler for CS-Library error and informational messages.	Set, retrieve, or clear.	If *action* is CS_SET, **buffer* is the message callback routine. If *action* is CS_GET, **buffer* is set to the address of the message callback routine that is currently installed. The default is NULL, which means no handler is installed.
CS_NOAPI_CHK	Whether or not CS-Library validates function arguments when library functions are called.	Set, retrieve, or clear.	CS_TRUE or CS_FALSE. CS_FALSE, the default, indicates that argument checking is performed.
CS_SYBASE_HOME	The location of an alternate Sybase home directory.	Set, retrieve, or clear.	A CS_CHAR string to the new Sybase home directory.
CS_USERDATA	User-allocated data.	Set, retrieve, or clear.	User-allocated data. A default is not applicable.

Value of property	Controls	Action	*buffer is
CS_VERSION	The version of CS-Library.	Retrieve only.	A symbolic code indicating the library version: • CS_VERSION_100 indicates the context exhibits version 10.0 behavior. • CS_VERSION_110 indicates version 11.1 behavior. • CS_VERSION_120 indicates the context exhibits version 12.0 behavior. • CS_VERSION_125 indicates version 12.5 behavior. • CS_VERSION_150 indicates version 15.0 behavior. • CS_VERSION_155 indicates version 15.5 behavior.

buffer

If a property value is being set, *buffer* points to the value to use in setting the property.

If a property value is being retrieved, *buffer* points to the space in which cs_config will place the value of the property.

If a property value is being cleared, pass *buffer* as NULL and *buflen* as CS_UNUSED.

buflen
Generally, *buflen* is the length, in bytes, of **buffer*.

If a property value is being set and the value in **buffer* is null-terminated, pass *buflen* as CS_NULLTERM.

If **buffer* is a fixed-length or symbolic value, pass *buflen* as CS_UNUSED.

outlen
A pointer to an integer variable.

outlen is not used if a property value is being set.

If a property value is being retrieved, cs_config sets **outlen* to the length, in bytes, of the requested information.

If the information is larger than *buflen* bytes, an application can use the value of **outlen* to determine how many bytes are needed to hold the information.

outlen can be passed as NULL if the application is setting a property value or does not require the output length of a retrieved value.

Return value

cs_config returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- There are three kinds of context properties:
  - Context properties specific to CS-Library
  - Context properties specific to Client-Library
  - Context properties specific to Server-Library

  cs_config sets and retrieves the values of CS-Library context properties. With the exception of CS_LOC_PROP, properties set using cs_config affect only CS-Library.

  ct_config sets and retrieves the values of Client-Library-specific context properties. Properties set using ct_config affect only Client-Library.

  srv_props sets and retrieves the values of Server-Library-specific context properties. Properties set using srv_props affect only Server-Library.
- See the "Properties" topics page in the *Open Client Client-Library/C Reference Manual* for information about Client-Library properties.

Application name property

- CS_APPNAME specifies the name that the application calls itself.
- cs_config sets the application name for a CS_CONTEXT structure. In a Client-Library application, allocated connections inherit the application name from their parent CS_CONTEXT structure.
- The application name specifies a section name in the Open Client and Open Server runtime configuration file. See "Runtime configuration file property" on page 19.
- CS_APPNAME cannot be set, retrieved, or cleared unless the CS_CONTEXT structure was allocated with CS_VERSION_110 or later.

Runtime configuration file property

- CS_CONFIG_FILE specifies the name and path to the Open Client and Open Server runtime configuration file.
- The default value is NULL, which means that the platform-specific default file will be used:
  - On UNIX platforms, the default file is *$SYBASE/$SYBASE_OCS/config/ocs.cfg*, where *$SYBASE* is the path to the Sybase installation directory; this path is specified as the value of the SYBASE environment variable.
  - On Windows platforms, the default file is *%SYBASE%\%SYBASE_OCS%\ini\ocs.cfg*, where *%SYBASE%* is the path to the Sybase installation directory; this path is specified as the value of the SYBASE environment variable.

  Note The default value may be overridden by the environment variable, SYBOCS_CFG.

- Setting the SYBOCS_CFG environment variable overrides the CS_EXTERNAL_CONFIG default. Note that this only affects applications which do not set the value of CSCONFIG_FILE via cs_config.
- If the default external-configuration file exists, Client-Library reads configuration settings from it unless the application explicitly sets the CS_EXTERNAL_CONFIG property to CS_FALSE. See "External configuration property" on page 20.
- CS_CONFIG_FILE cannot be set, retrieved, or cleared unless the CS_CONTEXT structure was allocated with CS_VERSION_110 or later.

External configuration property

- CS_EXTERNAL_CONFIG controls whether the Client-Library routine ct_init will read the Open Client and Open Server runtime configuration file to set default Client-Library property values for the CS_CONTEXT structure.
- The name of the Open Client and Open Server runtime configuration file is specified with the CS_CONFIG_FILE property. See "Runtime configuration file property" on page 19.
- The default for CS_EXTERNAL_CONFIG, CS_TRUE, depends on whether the default external-configuration file exists (see "Runtime configuration file property" on page 19). If the external-configuration file exists, then CS_EXTERNAL_CONFIG defaults to CS_TRUE. Otherwise, CS_EXTERNAL_CONFIG defaults to CS_FALSE.
- Configuration information is read from the section of the file labeled:

  ```
  [appname]
  ```

  where *appname* is the value of the CS_APPNAME property. (See "Application name property" on page 19.) If the application has not set the CS_APPNAME property, the configuration reads the section labeled:

  ```
  [DEFAULT]
  ```

- The "Using the Open Client and Open Server Runtime Configuration File" topics page in the *Open Client Client-Library/C Reference Manual* describes the syntax and keywords for configuration file entries.
- CS_EXTERNAL_CONFIG cannot be set, retrieved, or cleared unless the CS_CONTEXT structure is allocated with CS_VERSION_110 or later. (See cs_ctx_alloc.)

Extra Information property

- CS_EXTRA_INF determines whether or not CS-Library returns the extra information that is required to fill in a SQLCA, SQLCODE, or SQLSTATE structure.
- If an application is not retrieving messages into a SQLCA, SQLCODE, or SQLSTATE structure, the extra information is returned as ordinary CS-Library messages.

Locale information property

- The CS_LOC_PROP property defines a CS_LOCALE structure that contains localization information for a context. Localization information includes a language, character set, datetime, money, and numeric formats, and a collating sequence.
- CS_LOC_PROP affects both CS-Library and Client-Library, because a new connection picks up default localization information from its parent context.
- If an application does not call cs_config to define localization information for a context, the context uses default localization information that it picks up from the operating system environment when it is allocated. If localization information is not available in the operating system environment, the context uses platform-specific default localization values.
- The cs_loc_alloc routine allocates a CS_LOCALE structure.

CS-Library Message Callback property

- The CS_MESSAGE_CB property consists of a pointer to a user-supplied CS-Library message callback routine. The application uses this property to install a handler for error or informational messages from CS-Library.
  - The default value is NULL, meaning that no handler is installed.
  - An application function can be installed as a handler for CS-Library errors.
  - Once the handler is installed, CS-Library calls the handler when an error or exception occurs in a CS-Library routine.
- For a description and an example of coding a CS-Library error handler, see “Defining a CS-Library message callback” on page 6.
- The following code fragment demonstrates how a handler function is installed for CS-Library errors:

```
/*
** Install the function cslib_err_handler as the
** handler for CS-Library errors.
*/
if (cs_config(context, CS_SET, CS_MESSAGE_CB,
              (CS_VOID *)cslib_err_handler,
              CS_UNUSED, NULL)
    != CS_SUCCEED)
{
  /* Release the context structure.      */
```

```
        (void)cs_ctx_drop(context);
        fprintf(stdout,
          "Can't install CS-Lib error handler.\
          Exiting.\n");
        exit(1);
    }
```

- Client-Library applications that call CS-Library routines besides cs_ctx_alloc and cs_ctx_drop need dedicated CS-Library error handling. Applications should either install a CS-Library error handler or handle CS-Library errors inline with cs_diag.

  Note CS-Library error messages are not sent to the Client-Library error handler.

- Callback error handlers for Client-Library and CS-Library are installed differently:

  - An application installs Client-Library callback routines by calling ct_callback.
  - An application installs a CS-Library message callback routine by calling cs_config to set the value of the CS_MESSAGE_CB property.

  Aside from this difference, the CS-Library message callback is similar to the Client-Library client message callback. For general information on callback routines, see the "Callbacks" topics page in the *Open Client Client-Library/C Reference Manual*.

Argument checking for CS-Library calls

- The CS_NOAPI_CHK property determines whether or not CS-Library validates function arguments when a library function is called.
- If the value of CS_NOAPI_CHK is CS_FALSE (the default), then CS-Library checks arguments when API functions are called. Setting CS_NOAPI_CHK to CS_TRUE disables API checking.
- For argument checking, CS-Library validates the parameters passed with each function call. Pointers to CS-Library hidden structures such as CS_LOCALE are checked. Field values in structures are also checked for illegal combinations. If CS-Library finds invalid arguments and API checking is enabled, CS-Library generates error messages and the function fails. These messages can be trapped and displayed with a CS-Library callback error handler.

- If the value of CS_NOAPI_CHK is CS_TRUE, arguments are not validated before they are used. If the application passes invalid arguments to CS-Library, the application will not work right, resulting in memory corruption, memory access violations (UNIX "core dumps"), or incorrect results. No error messages are generated to warn the application of the condition. Do not disable API argument checking until the application has been completely debugged with API checking enabled.

  Warning! Do not set CS_NOAPI_CHK to CS_TRUE unless your application has been completely debugged with the default setting (CS_FALSE).

User-allocated data property

- The CS_USERDATA property defines user-allocated data. This property allows an application to associate user data with a particular context structure.
- CS-Library copies the user data into internal data space. An application can then call cs_config at a later time to retrieve the data.
- If you do not include a string's null terminator when calculating its length during the input stage, a call to cs_config (CS_GET) will return only the string (minus its null terminator). For example, if you input a 2-byte string with a null terminator, and specify the string's length as 2 bytes, cs_config (CS_GET) will return only the string. If, on the other hand, you input a 2-byte string with a null terminator and specify the string's length as 3 bytes, cs_config (CS_GET) will return the string and its null terminator.
- Although Client-Library also has a CS_USERDATA property, the Client-Library CS_USERDATA is set only at the connection and command levels.

Sybase home property

- The CS_SYBASE_HOME property specifies the name and path to an alternate Sybase home directory and overrides the environment variable *$SYBASE* (*%SYBASE%* for Windows).
- CS_SYBASE_HOME must be set before allocating a CS-Library context because the allocation of a context requires a valid Sybase home directory from which it will be set up. This means that CS_SYBASE_HOME must be set *before* calling cs_ctx_alloc() or cs_ctx_global(). cs_config() must be invoked with a NULL context to set CS_SYBASE_HOME. For example:

```
ret = cs_config(NULL, CS_SET, CS_SYBASE_HOME,
"/work/NewSybase", CS_NULLTERM, NULL);
```

You can also use the -y option of the isql and bcp utilities to specify an alternate Sybase home directory.

libtcl.cfg file property

- The CS_LIBTCL_CFG property specifies the name and path to an alternate *libtcl.cfg* file. As in the CS_SYBASE_HOME property, CS_LIBTCL_CFG is set by cs_config() using a NULL context and must be set before a CS-Library context is allocated. For example:

```
ret = cs_config(NULL, CS_SET, CS_LIBTCL_CFG,
"/work/Sybase/OCS-15_5/config/libtcl.cfg",
CS_NULLTERM, NULL);
```

Default interfaces file property

- The CS_DEFAULT_IFILE property specifies the name of an alternate *interfaces* file and its path. Unlike the CT-Library property CS_IFILE, CS_DEFAULT_IFILE does not override the use of alternate directory services that have already been specified in the *libtcl.cfg* file. The primary purpose of CS_DEFAULT_IFILE is to set a new default location for the *interfaces* file, in case the *interfaces* file is being used as the directory service.
- A CS-Library context must be allocated before calling cs_config() and it must be passed in cs_config() while setting the CS_DEFAULT_IFILE property. For example:

```
ret = cs_config(ctx, CS_SET, CS_DEFAULT_IFILE,
"/work/NewSybase/interfaces", CS_NULLTERM, NULL);
```

Version level property

- The CS_VERSION property represents the version of CS-Library behavior that an application has requested using cs_ctx_alloc.
- An application can only retrieve the value of CS_VERSION.
- Possible values for CS_VERSION include the following:
  - CS_VERSION_100 indicates version 10.0 behavior
  - CS_VERSION_110 indicates version 11.1 behavior
  - CS_VERSION_120 indicates version 12.0 behavior
  - CS_VERSION_125 indicates version 12.5 behavior
  - CS_VERSION_150 indicates version 15.0 behavior.
  - CS_VERSION_155 indicates version 15.5 behavior.

See also

cs_ctx_alloc, ct_con_props, ct_config, ct_init

cs_conv_mult

Description

Retrieves the conversion multiplier for converting character data from one character set to another.

Syntax

```
CS_RETCODE cs_conv_mult(context,
                        srcloc,
                        destloc,
                        conv_multiplier)
CS_CONTEXT      *context;
CS_LOCALE       *srcloc;
CS_LOCALE       *destloc;
CS_INT           *conv_multiplier;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

srcloc

A pointer to the CS_LOCALE structure that describes the source variable's character set. This parameter cannot be NULL.

destloc

A pointer to the CS_LOCALE structure that describes the destination variable's character set. This parameter cannot be NULL.

conv_multiplier

A pointer to a CS_INT variable. cs_conv_mult retrieves the conversion multiplier for conversions from the character set indicated by *srcloc* to the character set indicated by *destloc* and places it into **conv_multiplier*.

Return value

cs_conv_mult returns the following values:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

The most common reason for a cs_conv_mult failure is an invalid parameter.

Examples

The following code fragment retrieves the conversion multiplier for conversions from the iso_1 character set to the eucjis character set:

```
#define EXIT_ON_FAIL(context, ret, msg) \
{ if (ret != CS_SUCCEED) \
    { \
      fprintf(stdout,"Fatal error(%ld): %s\n",(long)ret,msg); \
      if (context != (CS_CONTEXT *)NULL) \
      { (CS_VOID)ct_exit(context, CS_FORCE_EXIT); \
        (CS_VOID)cs_ctx_drop(context); } \
      exit(-1); \
```

```
    } }

** usa_locale uses the iso_1 character set.
*/
ret = cs_loc_alloc(context, &usa_locale);
EXIT_ON_FAIL(context, ret, "cs_loc_alloc(usa) failed.");
ret = cs_locale(context, CS_SET, usa_locale,
                CS_SYB_CHARSET, "iso_1", CS_NULLTERM, NULL);
EXIT_ON_FAIL(context, ret, "cs_locale(usa, CHARSET) failed.");

/*
 ** japan_locale uses eucjis.
 */
 ret = cs_loc_alloc(context, &japan_locale);
 EXIT_ON_FAIL(context, ret, "cs_loc_alloc(japan) failed.");

ret = cs_locale(context, CS_SET, japan_locale,
                 CS_SYB_CHARSET, "eucjis", CS_NULLTERM, NULL);
 EXIT_ON_FAIL(context, ret, "cs_locale(japan, CHARSET) failed.");

/*
 ** Get the conversion multiplier for iso_1 to eucjis conversions.
 */
 ret = cs_conv_mult(context,
                    usa_locale, japan_locale, &conv_mult);
 EXIT_ON_FAIL(context, ret, "cs_conv_mult(usa, japan) failed.");

fprintf(stdout,
         "Conversion multiplier for iso_1 to eucjis is %ld.\n",
         (long)conv_mult);
```

Usage

- cs_conv_mult retrieves the conversion multiplier for converting character data from one character set to another.
- The conversion multiplier allows an application to size the destination data space for conversion of character data into a different character set.
- When converted to another character set, character strings can grow or shrink in length, and applications need to make sure that the destination data space is large enough for the result. With a multi-byte character set destination, 1-byte in the source can convert to several bytes in the destination.
- Inconvertible characters are substituted with single-byte “?”, 2-byte “??” or 3-byte “0xefbfbd” characters.
- A *conversion multiplier* equals the largest number of bytes in the destination that can replace 1 source byte.

- When converting a character string to a different character set, the application uses the conversion multiplier to size the destination data space, as follows:

```
bytes_needed = conv_mult
               * srclen
               * CS_SIZEOF(CS_BYTE)
               + NTB
```

  where:

  - *bytes_needed* is the necessary length, in bytes, of the destination data space.
  - *conv_mult* is the conversion multiplier value.
  - *srclen* is the length, in bytes, of the source string.
  - *NTB* is 1 if null termination is requested and 0 otherwise.

- See the *Open Client and Open Server International Developers Guide*.

See also cs_convert, cs_locale, cs_manage_convert

cs_convert

Description Converts a data value from one datatype, locale, or format to another datatype, locale, or format.

Syntax

```
CS_RETCODE cs_convert(context, srcfmt, srcdata,
                      destfmt, destdata, resultlen)
CS_CONTEXT      *context;
CS_DATAFMT      *srcfmt;
CS_VOID         *srcdata;
CS_DATAFMT      *destfmt;
CS_VOID         *destdata;
CS_INT          *resultlen;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

srcfmt

A pointer to a CS_DATAFMT structure describing the source data format. The fields in **srcfmt* are used as follows:

Field name	Set it to
datatype	A type constant representing the type of the source data (CS_CHAR_TYPE, CS_BINARY_TYPE, and so on).
maxlength	The length of the data in the **srcdata* buffer. This value is ignored for fixed-length datatypes or if *srcdata* is NULL.
locale	A pointer to a CS_LOCALE structure containing localization values for the source data, or NULL to use localization values from **context*.

All other fields are ignored.

For general information on the CS_DATAFMT structure, see the "CS_DATAFMT Structure" topics page in the *Open Client Client-Library/C Reference Manual*.

srcdata

A pointer to the source data. To indicate that the source data represents a null value, pass *srcdata* as NULL. If *srcdata* is NULL, cs_convert places the null substitution value for the datatype indicated by *destfmt–>datatype* in **destdata*.

Table 2-17 on page 88 lists the default null substitution value for each datatype. An application can define custom null substitution values by calling cs_setnull.

destfmt

A pointer to a CS_DATAFMT structure describing the destination data format. Table 2-2 lists the fields in **destfmt* that are used.

Table 2-2: CS_DATAFMT fields for cs_convert's *destfmt parameter

Field Name	Set it to
datatype	A type constant representing the desired destination datatype (CS_CHAR_TYPE, CS_BINARY_TYPE, and so on).
maxlength	The length of the *destdata* buffer.
locale	A pointer to a CS_LOCALE structure containing localization values for the destination data, or NULL to use localization values from **context*.
format	A bit mask of the following symbols: • For character and text destinations only, use CS_FMT_NULLTERM to null-terminate the data, or CS_FMT_PADBLANK to pad to the full length of the variable with spaces. • For character, binary, text, and image destinations, use CS_FMT_PADNULL to pad to the full length of the variable with nulls. • When converting from a character source to a character destination, use CS_FMT_SAFESTR to double any occurrences of the single-quote character (') in the destination. CS_FMT_SAFESTR can be combined with CS_FMT_NULLTERM, CS_FMT_PADBLANK, or CS_FMT_PADNULL. • For any type of destination, use CS_FMT_UNUSED if no format information is being provided.
scale	The scale used for the destination variable. If the source data is the same type as the destination, *scale* can be set to CS_SRC_VALUE to indicate that the destination should pick up its value for *scale* from the source data. *scale* must be less than or equal to *precision*.
precision	The precision used for the destination variable. If the source data is the same type as the destination, *precision* can be set to CS_SRC_VALUE to indicate that the destination should pick up its value for *precision* from the source data. *precision* must be greater than or equal to *scale*.

All other fields are ignored.

For general information on the CS_DATAFMT structure, see the "CS_DATAFMT Structure" topics page in the *Open Client Client-Library/C Reference Manual*.

destdata

A pointer to the destination buffer space.

resultlen

A pointer to an integer variable. cs_convert sets **resultlen* to the length, in bytes, of the data placed in **destdata*. If the conversion fails, cs_convert sets **resultlen* to CS_UNUSED.

resultlen is an optional parameter and can be passed as NULL.

Datatype Conversion Chart

Table 2-3 indicates which datatype conversions are supported by cs_convert. The source datatypes are listed in the leftmost column and the destination datatypes are listed in the top row of the chart. "•" indicates that the conversion is supported; a blank space indicates that the conversion is not supported.

Table 2-3: Datatype conversion chart

Open Client datatypes	CS_BINARY	CS_LONGBINARY	CS_VARBINARY	CS_BIT	CS_CHAR	CS_LONGCHAR	CS_VARCHAR	CS_DATETIME	CS_DATETIME4	CS_TINYINT	CS_SMALLINT	CS_INT	CS_DECIMAL	CS_NUMERIC	CS_FLOAT	CS_REAL	CS_MONEY	CS_MONEY4	CS_BOUNDARY	CS_SENSITIVITY	CS_TEXT	CS_IMAGE	CS_UNICHAR	CS_DATE	CS_TIME	CS_BIGINT	CS_USMALLINT	CS_UINT	CS_UBIGINT	CS_UNITEXT	CS_XML
CS_BINARY	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	•	•		•
CS_LONGBINARY	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	•	•		•
CS_VARBINARY	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	•	•		•
CS_BIT	•	•	•	•	•	•	•			•	•	•	•	•	•	•					•	•	•				•	•	•		
CS_CHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	•		•
CS_LONGCHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	•		•
CS_VARCHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	•		•
CS_DATETIME			•		•	•	•	•	•												•		•	•	•						
CS_DATETIME4			•		•	•	•	•	•												•		•	•	•						
CS_TINYINT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_SMALLINT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_INT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_DECIMAL	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_NUMERIC	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_FLOAT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_REAL	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_MONEY	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_MONEY4	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•				•	•	•		
CS_BOUNDARY					•	•	•												•		•										

Open Client datatypes	CS_BINARY	CS_LONGBINARY	CS_VARBINARY	CS_BIT	CS_CHAR	CS_LONGCHAR	CS_VARCHAR	CS_DATETIME	CS_DATETIME4	CS_TINYINT	CS_SMALLINT	CS_INT	CS_DECIMAL	CS_NUMERIC	CS_FLOAT	CS_REAL	CS_MONEY	CS_MONEY4	CS_BOUNDARY	CS_SENSITIVITY	CS_TEXT	CS_IMAGE	CS_UNICHAR	CS_DATE	CS_TIME	CS_BIGINT	CS_USMALLINT	CS_UINT	CS_UBIGINT	CS_UNITEXT	CS_XML
CS_SENSITIVITY					•	•	•														•	•									
CS_TEXT	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	•		•
CS_IMAGE	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	•	•	•	•
CS_UNICHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	•	•		
CS_DATE			•		•	•	•	•	•												•		•	•							
CS_TIME			•		•	•	•	•	•												•		•		•						
CS_BIGINT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•			•	•	•	•		
CS_USMALLINT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•			•		•	•		
CS_UINT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•			•	•		•		
CS_UBIGINT	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•			•	•	•			•	•	•			
CS_UNITEXT	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•	•	•	•
CS_XML	•	•	•		•	•	•														•	•									

Datetime Datatypes Conversion Chart

Table 2-4 indicates which datatype conversions are supported by cs_convert for the datetime datatypes. The source datatypes are listed in the leftmost column, and the destination datatypes are listed in the top row of the chart. "•" indicates that the conversion is supported. A blank space indicates that the conversion is not supported.

Table 2-4: Datetime datatype conversions

Open Client datatypes	CS_BINARY	CS_LONGBINARY	CS_VARBINARY	CS_BIT	CS_CHAR	CS_LONGCHAR	CS_VARCHAR	CS_DATETIME	CS_DATETIME4	CS_BIGDATETIME	CS_TINYINT	CS_SMALLINT	CS_INT	CS_DECIMAL	CS_NUMERIC	CS_FLOAT	CS_REAL	CS_MONEY	CS_MONEY4	CS_BOUNDARY	CS_SENSITIVITY	CS_TEXT	CS_IMAGE	CS_UNICHAR	CS_DATE	CS_TIME	CS_BIGINT	CS_BIGTIME	CS_UNITEXT
CS_BINARY	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	
CS_LONGBINARY	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	
CS_VARBINARY	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	
CS_BIT	•	•	•	•	•	•	•				•	•	•	•	•	•	•					•	•	•					
CS_CHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	

Open Client datatypes	CS_BINARY	CS_LONGBINARY	CS_VARBINARY	CS_BIT	CS_CHAR	CS_LONGCHAR	CS_VARCHAR	CS_DATETIME	CS_DATETIME4	CS_BIGDATETIME	CS_TINYINT	CS_SMALLINT	CS_INT	CS_DECIMAL	CS_NUMERIC	CS_FLOAT	CS_REAL	CS_MONEY	CS_MONEY4	CS_BOUNDARY	CS_SENSITIVITY	CS_TEXT	CS_IMAGE	CS_UNICHAR	CS_DATE	CS_TIME	CS_BIGINT	CS_BIGTIME	CS_UNITEXT
CS_LONGCHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	
CS_VARCHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	
CS_DATETIME			•		•	•	•	•	•	•												•		•	•	•		•	
CS_DATETIME4			•		•	•	•	•	•	•												•		•	•	•		•	
CS_BIGDATETIME			•		•	•	•	•	•	•												•		•	•	•		•	•
CS_TINYINT	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_SMALLINT	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_INT	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_DECIMAL	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_NUMERIC	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_FLOAT	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_REAL	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_MONEY	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_MONEY4	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•					
CS_BOUNDARY					•	•	•													•		•							
CS_SENSITIVITY					•	•	•														•	•							
CS_TEXT	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	
CS_IMAGE	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•			•
CS_UNICHAR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•		•	
CS_DATE			•		•	•	•	•	•	•												•		•	•				
CS_TIME			•		•	•	•	•	•	•												•		•		•		•	
CS_BIGINT	•	•	•	•	•	•	•				•	•	•	•	•	•	•	•	•			•	•	•			•		
CS_BIGTIME			•		•	•	•	•	•	•												•		•		•		•	•
CS_UNITEXT	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	•	•	•	•	•

Conversion between CS_BIGDATETIME and CS_BIGTIME and the following datatypes is not supported:

- CS_BLOB
- CS_LONG
- CS_UBIGINT
- CS_UINT
- CS_USHORT
- CS_USMALLINT
- CS_XML

All conversions to and from CS_BIGDATETIME and CS_BIGTIME are handled in the same way as existing datetime and time conversions.

Return value

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

A common reason for a cs_convert failure is that CS-Library does not support the requested conversion.

cs_convert conversion errors generate CS-Library error messages. See "Error handling" on page 4.

Usage

- To determine whether a particular conversion is permitted, use cs_will_convert.
- In Client-Library applications, ct_bind sets up automatic, implicit data conversion, which makes it unnecessary for an application to explicitly convert result data that is bound to program variables.
- An application can install custom conversion routines by calling cs_set_convert. Once a custom routine for a particular type of conversion is installed, cs_convert or ct_bind call the custom routine whenever a conversion of that type is required.
- cs_convert can convert between standard and user-defined datatypes. To enable these types of conversions, an application must install the appropriate custom conversion routines using cs_set_convert.
- See the "Types" topics page in the *Open Client Client-Library/C Reference Manual*. For information about Adaptive Server Enterprise datatypes, see the *Adaptive Server Enterprise Reference Manual*.

About specific conversions

- A conversion to or from *binary* and *image* datatypes is a straight byte-copy, except when the conversion involves *character* or *text* data. When converting character or text data to binary or image, cs_convert interprets the character or text string as hexadecimal, whether or not the string contains a leading "0x." There must be a match in the lengths of binary data and fixed length data. If the data lengths do not match, there will be underflow or overflow.
- Converting a *money*, *character*, or *text* value to *float* can result in a loss of precision. Converting a float value to character or text can also result in a loss of precision.
- Any length mismatch in the conversion to and from binary and image datatypes cause error *underflow* or *overflow*. This may happen, for example, if you are converting 1-byte binary data to integer data. Use datatype CS_TINYINT (1-byte integer) to avoid an error.
- Converting a *float* value to *money* can result in overflow, because the maximum CS_MONEY value is $922,337,203,685,477.5807, and the maximum CS_MONEY4 value is $214,748.3648.
- If overflow occurs when converting *integer* or *float* data to *character* or *text*, the first character of the resulting value will contain an asterisk (*) to indicate the error.
- A conversion to *bit* has the following effect: If the value being converted is not 0, the bit value is set to 1; if the value is 0, the bit value is set to 0.
- When converting *decimal* or *numeric* data to decimal or numeric data, CS_SRC_VALUE can be used in *destfmt–>scale* and *destfmt–>precision* to indicate that the destination data should have the same scale and precision as the source. CS_SRC_VALUE is valid only when the source data is decimal or numeric.

Note Open Client and Open Server 15.0 and later support the unichar datatype. For information about this datatype, see Chapter 3, "Using Open Client and Open Server Datatypes", in the *Open Client Client-Library/C Programmers Guide*.

Converting between character sets

- cs_convert performs character set conversion when:
  - *srctype* and *desttype* both represent character-based types, and

- *srcfmt–>locale* specifies a different character set than *destfmt–>locale*.

The character-based types are CS_CHAR, CS_LONGCHAR, CS_TEXT, CS_UNITEXT, CS_VARCHAR or CS_XML.

- You can program an application to perform character-set conversion by following these steps:

  a Call cs_loc_alloc twice to allocate two CS_LOCALE structures, *src_locale* and *dest_locale*, which will be configured to describe the locale of the source data and destination data, respectively.

  b Configure the character set associated with *src_locale* by calling cs_locale. The call can specify either a locale name or a character set name.

  To use a character set name, pass *action* as CS_SET, *type* as CS_SYB_CHARSET, and *buffer* as the name of the character set. Repeat to configure the character set for *dest_locale*.

  To use a locale name, pass *action* as CS_SET, *type* as CS_LC_CTYPE, and *buffer* as a locale name (the character set associated with the locale name will be used). Repeat to configure the character set for *dest_locale*.

  c (Optional) Call cs_conv_mult to get the conversion multiplier for conversions between *src_locale*'s character set and *dest_locale*'s character set. The conversion multiplier can be used to determine whether the result can possibly overflow the destination space.

  d Configure the CS_DATAFMT structures to describe the datatype, length, and format of the source and destination data. Set the *locale* field in the source CS_DATAFMT structure to *src_locale*, and set the locale field in the destination CS_DATAFMT structure to *dest_locale*.

  e Call cs_convert to perform the conversion. This step can be repeated as many times as necessary, using the configured CS_LOCALE and CS_DATAFMT structures.

  f Call cs_loc_drop to deallocate each of *src_locale* and *dest_locale* when they are no longer needed.

See also

cs_conv_mult, cs_manage_convert, cs_set_convert, cs_setnull, cs_will_convert

cs_ctx_alloc

Description

Allocates a CS_CONTEXT structure.

Syntax

```
CS_RETCODE cs_ctx_alloc(version, ctx_pointer)

CS_INT          version;
CS_CONTEXT      **ctx_pointer;
```

Parameters

version

One of the following symbolic values, to indicate the intended version of CS-Library behavior:

Value of version	Indicates	Features supported
CS_VERSION_100	10.0 behavior	Initial version.
CS_VERSION_110	11.1 behavior	Unicode character set support. Use of external configuration files to control Client-Library property settings.
CS_VERSION_120	12.0 behavior	All above features.
CS_VERSION_125	12.5 behavior	unichar support, wide data and columns, SSL.
CS_VERSION_150	15.0 behavior	BCP partitions, BCP computed columns, large identifiers, Unilib®, Adaptive Server Enterprise default packet size, scrollable cursors, and clusters support. Also, support for unitext, xml, bigint, usmallint, uint, and ubigint datatypes. Note the Sybase library name change.
CS_VERSION_155	15.5 behavior	CS_BIGDATETIME and CS_BIGTIME datatypes and microsecond granularity for time data, ct_send_data enhancement, Open Server dynamic listeners, Open Client CS_RES_ NOXNLMETADATA response capability, FIPS-140-2-compliant password encryption.

ctx_pointer

The address of a pointer variable. cs_ctx_alloc sets **ctx_pointer* to the address of a newly allocated CS_CONTEXT structure.

In case of error, cs_ctx_alloc sets **ctx_pointer* to NULL.

Return value

cs_ctx_alloc returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_MEM_ERROR	The routine failed because it could not allocate sufficient memory.
CS_FAIL	The routine failed for other reasons.

The most common reason for a cs_ctx_alloc failure is a misconfigured system environment. cs_ctx_alloc must read the locales file that specifies the default localization values for the allocated context. If CS-Library cannot find the locales file or if the locales file is misconfigured, cs_ctx_alloc fails.

Note When cs_ctx_alloc returns CS_FAIL an extended error message is sent to standard error (SDTERR) and to the *sybinit.err* file that is created in the current working directory.

On most systems, the SYBASE environment variable or logical name specifies the location of the locales file. The *Open Client and Open Server Configuration Guide for Microsoft Windows* or *Open Client and Open Server Configuration Guide for UNIX* describes the environmental configuration required for CS-Library localization values.

Other common reasons for a cs_ctx_alloc failure include:

- Memory is insufficient.
- Localization files are missing.
- The value of the LANG environment variable does not match an entry in the locales file.

Note On platforms that have a standard error device, cs_ctx_alloc prints U.S. English error messages to the standard error device when CS-Library cannot find the locales file. For Windows and other platforms that lack a standard error device, U.S. English error messages are written to a text file called *sybinit.err* in the application's working directory.

Examples

```
/*
 ** ex_init()
 */

CS_RETCODE CS_PUBLIC
 ex_init(context)
 CS_CONTEXT*     *context;
 {
      CS_RETCODE         retcode;
      CS_INT             netio_type = CS_SYNC_IO;

     /* Get a context handle to use */
      retcode = cs_ctx_alloc(CS_VERSION_125, context);
      if (retcode != CS_SUCCEED)
      {
           ex_error("ex_init: cs_ctx_alloc() failed");
           return retcode;
      }

      /* Initialize Open Client */
       ...CODE DELETED.....

      /* Install client and server message handlers */
       ...CODE DELETED.....

      if (retcode != CS_SUCCEED)
       {
           ct_exit(*context, CS_FORCE_EXIT);
           cs_ctx_drop(*context);
           *context = NULL;
       }

      return retcode;
}
```

Usage

- A CS_CONTEXT structure, also called a “context structure,” contains information that describes an application context. For example, a context structure contains default localization information and defines the version of CS-Library that is in use.
- Allocating a context structure is the first step in any Client-Library or Server-Library application.
- After allocating a CS_CONTEXT structure, a Client-Library application typically customizes the context by calling cs_config and/or ct_config to create one or more connections within the context. A Server-Library application can customize a context by calling cs_config and srv_props.

- To deallocate a context structure, an application can call cs_ctx_drop.
- cs_ctx_global also allocates a context structure. The difference between cs_ctx_alloc and cs_ctx_global is that cs_ctx_alloc allocates a new context structure each time it is called, while cs_ctx_global allocates a new context structure only once, the first time it is called. On subsequent calls, cs_ctx_global simply returns a pointer to the existing context structure.
- To allow the use of sigwait in signal handling for multithreaded applications, both cs_ctx_alloc and cs_ctx_global will block signals the first time they are executed for a multi-threaded application. All these signals are blocked except for one dedicated thread, which is controlled by the Open Client/Open Server libraries. This thread will block the signal when a corresponding signal handler is installed using the ct_callback or srv_signal routines. A separate thread is subsequently spawned to invoke sigwait for this signal and to execute the appropriate user-provided signal handler function when the signal is received. For information on how to override this behavior and allow your application to handle thread signals itself, see the Chapter 2, "Client-Library Topics" in the *Open Client Client-Library/C Reference Manual*.

See also

ct_con_alloc, ct_config, cs_ctx_drop, cs_ctx_global, cs_config

cs_ctx_drop

Description

Deallocates a CS_CONTEXT structure.

Syntax

```
CS_RETCODE cs_ctx_drop(context)

 CS_CONTEXT     *context;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

Return value

cs_cxt_drop returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

cs_ctx_drop returns CS_FAIL if the context contains an open connection.

Examples

```
/*
```

```
** ex_ctx_cleanup()
**
** Parameters:
**    context      Pointer to context structure.
**    status       Status of last interaction with Client-
**                 Library. If not ok, this routine will perform
**                 a force exit.
**
** Returns:
**    Result of function calls from Client-Library.
     */
     CS_RETCODE CS_PUBLIC
     ex_ctx_cleanup(context, status)
     CS_CONTEXT         *context;
     CS_RETCODE          status;
     {
        CS_RETCODE         retcode;
        CS_INT             exit_option;
        exit_option = (status != CS_SUCCEED) ? CS_FORCE_EXIT :
           CS_UNUSED;
        retcode = ct_exit(context, exit_option);
        if (retcode != CS_SUCCEED)
        {
           ex_error("ex_ctx_cleanup: ct_exit() failed");
           return retcode;
        }
        retcode = cs_ctx_drop(context);
        if (retcode != CS_SUCCEED)
        {
           ex_error("ex_ctx_cleanup: cs_ctx_drop() failed");
           return retcode;
        }
        return retcode;
     }
```

Usage

- A CS_CONTEXT structure describes a particular context, or operating environment, for a set of server connections.
- Once a CS_CONTEXT has been deallocated, it cannot be used again. To allocate a new CS_CONTEXT, an application can call cs_ctx_alloc.

Note Sybase supports only one context handler per application program.

- A Client-Library application cannot call cs_ctx_drop to deallocate a CS_CONTEXT structure until it has called ct_exit to clean up Client-Library space associated with the context.

See also cs_ctx_alloc, ct_close, ct_exit

cs_ctx_global

Description Allocates or returns a CS_CONTEXT structure.

Syntax CS_RETCODE cs_ctx_global(version, ctx_pointer)

```
CS_INT          version;
CS_CONTEXT   **ctx_pointer;
```

Parameters

version

One of the following symbolic values, to indicate the intended version of CS-Library behavior:

Value of version	Indicates	Features supported
CS_VERSION_100	10.0 behavior	Initial version.
CS_VERSION_110	11.1 behavior	Unicode character set support. Use of external configuration files to control Client-Library property settings.
CS_VERSION_120	12.0 behavior	All above features.
CS_VERSION_125	12.5 behavior	unichar support, wide data and columns, SSL.
CS_VERSION_150	15.0 behavior	BCP partitions, BCP computed columns, large identifiers, Unilib, Adaptive Server Enterprise default packet size, scrollable cursors, and clusters support. Also support for unitext, xml, bigint, usmallint, uint, and ubigint datatypes. Note Sybase library name change.
CS_VERSION_155	15.5 behavior	CS_BIGDATETIME and CS_BIGTIME datatypes and microsecond granularity for time data, ct_send_data enhancement, Open Server dynamic listeners, Open Client CS_RES_ NOXNLMETADATA response capability, FIPS-140-2-compliant password encryption.

If an application has already allocated a CS_CONTEXT structure, *version* must match the version previously requested.

ctx_pointer

The address of a pointer variable. cs_ctx_global sets **ctx_pointer* to the address of a new or previously allocated CS_CONTEXT structure.

In case of error, cs_ctx_global sets **ctx_pointer* to NULL.

Return value

cs_ctx_global returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_MEM_ERROR	The routine failed because it could not allocate sufficient memory.
CS_FAIL	The routine failed for other reasons.

Common reasons for a cs_ctx_global failure include:

- A lack of available memory
- A *version* value that does not match a previously requested version

Note When cs_ctx_global returns CS_FAIL an extended error message is sent to standard error (SDTERR) and to the *sybinit.err* file that is created in the current working directory.

The first cs_ctx_global call to execute in an application can fail due to configuration problems. See "Returns" under cs_ctx_alloc in this chapter.

Usage

- cs_ctx_alloc also allocates a context structure. The only difference between cs_ctx_alloc and cs_ctx_global is that cs_ctx_alloc allocates a new context structure each time it is called, while cs_ctx_global allocates a new context structure only once, the first time it is called. On subsequent calls, cs_ctx_global simply returns a pointer to the existing context structure.
- cs_ctx_global is of use in applications that need to access a single context structure from multiple independent modules.
- To allow the use of sigwait in signal handling for multithreaded applications, both cs_ctx_alloc and cs_ctx_global will block signals the first time they are executed for a multi-threaded application. All these signals are blocked except for one dedicated thread, which is controlled by the Open Client/Open Server libraries. This thread will block the signal when a corresponding signal handler is installed using the ct_callback or srv_signal routines. A separate thread is subsequently spawned to invoke sigwait for this signal and to execute the appropriate user-provided signal handler function when the signal is received. For information on how to override this behavior and allow your application to handle thread signals itself, see the Chapter 2, "Client-Library Topics" in the *Open Client Client-Library/C Reference Manual*.
- See cs_ctx_alloc in this chapter.

See also

cs_ctx_alloc, cs_ctx_drop, cs_config, ct_con_alloc, ct_config

cs_diag

Description

Manages inline error handling.

Syntax

```
CS_RETCODE cs_diag(context, operation, type, index,
        buffer)

CS_CONTEXT   *context;
CS_INT       operation;
CS_INT       type;
CS_INT       index;
CS_VOID       *buffer;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

operation

The operation to perform. Table 2-5 on page 45 lists the legal symbolic values for *operation*.

type

Depending on the value of *operation*, *type* indicates either the type of structure to receive message information or the type of message on which to operate, or both.

Possible values are:

Value of type	Indicates
SQLCA_TYPE	A SQLCA structure.
SQLCODE_TYPE	A SQLCODE structure, which is a long integer.
SQLSTATE_TYPE	A SQLSTATE structure, which is a 6-element character array.
CS_CLIENTMSG_TYPE	A CS_CLIENTMSG structure. Also used to indicate CS-Library messages.

index

The index of the message of interest. The first message has an index of 1, the second an index of 2, and so forth.

buffer

A pointer to data space.

Depending on the value of *operation*, buffer can point to a structure or a CS_INT.

Return value

cs_diag returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_NOMSG	The application attempted to retrieve a message whose index is higher than the highest valid index. For example, the application attempted to retrieve message number 3 but only 2 messages were available.

Common reasons for a cs_diag failure include:

- Invalid *context*
- Inability to allocate memory
- Invalid parameter combination

Usage

Table 2-5: Summary of cs_diag parameter usage

Value of operation	cs_diag	type is	index is	buffer is
CS_INIT	Initializes inline error handling.	CS_UNUSED	CS_UNUSED	NULL
CS_MSGLIMIT	Sets the maximum number of messages to store.	CS_CLIENTMSG_ TYPE	CS_UNUSED	A pointer to an integer value.
CS_CLEAR	Clears message information for this context. If *buffer* is not NULL, cs_diag also clears the **buffer* structure by initializing it with blanks and/or NULLs, as appropriate.	One of the legal *type* values.	CS_UNUSED	A pointer to a structure whose type is defined by *type*, or NULL.
CS_GET	Retrieves a specific message.	One of the legal *type* values.	The one-based index of the message to retrieve.	A pointer to a structure whose type is defined by *type*.

Value of operation	cs_diag	type is	index is	buffer is
CS_STATUS	Returns the current number of stored messages.	CS_CLIENTMSG_TYPE	CS_UNUSED	A pointer to an integer value.

- An application that includes calls to CS-Library can handle CS-Library messages in one of two ways:
  - The application can call cs_config to install a CS-Library message callback, or
  - The application can handle CS-Library messages inline, using cs_diag.

  An application can switch back and forth between the inline method and the callback method:
  - Installing a CS-Library message callback turns off inline message handling. Any saved messages are discarded.
  - Likewise, cs_diag(CS_INIT) "de-installs" an application's CS-Library message callback. If the application has a message callback installed when cs_diag(CS_INIT) is called, the application's first CS_GET call to cs_diag will retrieve a warning message to this effect.

  If a CS-Library message callback is not installed and inline message handling is not enabled, CS-Library discards message information.
- cs_diag manages inline message handling for a specific context. If an application has more than one context, it must make separate cs_diag calls for each context.
- In a multithreaded application, cs_diag reports only those messages that pertain to CS-Library calls made by the thread which has called cs_diag. See the "Multithreaded Programming" topics page in the *Open Client Client-Library/C Reference Manual*.
- cs_diag allows an application to retrieve message information into a CS_CLIENTMSG structure or a SQLCA, SQLCODE, or SQLSTATE structure. When retrieving messages, cs_diag assumes that *buffer* points to a structure of the type indicated by *type*.

An application that is retrieving messages into a SQLCA, SQLCODE, or SQLSTATE structure must set the CS-Library context property CS_EXTRA_INF to CS_TRUE. This is because the SQL structures contain information that is not ordinarily returned by CS-Library's error-handling mechanism.

An application that is not using the SQL structures can also set CS_EXTRA_INF to CS_TRUE. In this case, the extra information is returned as standard CS-Library messages.

- If cs_diag does not have sufficient internal storage space in which to save a new message, it throws away all unread messages and stops saving messages. The next time it is called with *operation* as CS_GET, it returns a special message to indicate the space problem.

  After returning this message, cs_diag starts saving messages again.

Initializing inline error handling

- To initialize inline error handling, an application calls cs_diag with *operation* as CS_INIT.
- Generally, if a context will use inline error handling, the application should call cs_diag to initialize inline error handling for the context immediately after allocating it.

Clearing messages

- To clear message information for a context, an application calls cs_diag with *operation* as CS_CLEAR.
  - cs_diag assumes that *buffer* points to a structure whose datatype corresponds to the value of *type*.
  - cs_diag clears the **buffer* structure by setting it to blanks and/or NULLs, as appropriate.
- Message information is not cleared until an application explicitly calls cs_diag with *operation* as CS_CLEAR. Retrieving a message does not remove it from the message queue.

Retrieving messages

- To retrieve message information, an application calls cs_diag with *operation* as CS_GET, *type* as the type of structure in which to retrieve the message, *index* as the one-based index of the message of interest, and **buffer* as a structure of the appropriate type.
- cs_diag fills in the **buffer* structure with the message information.

- If an application attempts to retrieve a message whose index is higher than the highest valid index, cs_diag returns CS_NOMSG to indicate that no message is available.
- See the "SQLCA Structure," "SQLCODE Structures," "SQLSTATE structure," and "CS_CLIENTMSG Structure" topics pages in the *Open Client Client-Library/C Reference Manual* for information on these structures.

Limiting messages

- If an application runs on platforms with limited memory, you may want to limit the number of messages that CS-Library saves in that application.
- To limit the number of saved messages, an application calls cs_diag with *operation* as CS_MSGLIMIT and *type* as CS_CLIENTMSG_TYPE.
- When a message limit is reached, CS-Library discards any new messages.
- An application cannot set a message limit that is less than the number of messages currently saved.
- CS-Library's default behavior is to save an unlimited number of messages. An application can restore this default behavior by setting a message limit of CS_NO_LIMIT.

Retrieving the number of messages

- To retrieve the number of current messages, an application calls cs_diag with *operation* as CS_STATUS and *type* as the CS_CLIENTMSG_TYPE.

See also

ct_callback, ct_options

cs_dt_crack

Description

Converts a machine-readable datetime value into a user-accessible format.

Syntax

```
CS_RETCODE cs_dt_crack(context, datetype, dateval,
                daterec)

CS_CONTEXT      *context;
CS_INT          datetype;
CS_VOID         *dateval;
CS_DATEREC      *daterec;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

datetype

A symbolic value indicating the datatype of **dateval*:

Value of datetype	Indicates
CS_DATE_TYPE	CS_DATE **dateval*.
CS_TIME_TYPE	CS_TIME **dateval*
CS_DATETIME_TYPE	A CS_DATETIME **dateval*.
CS_DATETIME4_TYPE	A CS_DATETIME4 **dateval*.
CS_BIGDATETIME_TYPE	CS_BIGDATETIME **dateval*.
CS_BIGTIME_TYPE	CS_BIGTIME **dateval*

dateval

A pointer to the date, time, or datetime value to be converted.

daterec

A pointer to a CS_DATEREC structure. cs_dt_crack fills this structure with the translated date, time, or datetime value. A CS_DATEREC is defined as follows:

```
typedef struct cs_daterec {
     CS_INT      dateyear;      /* year              */
     CS_INT      datemonth;     /* month             */
     CS_INT      datedmonth;    /* day of month      */
     CS_INT      datedyear;     /* day of year       */
     CS_INT      datedweek;     /* day of week       */
     CS_INT      datehour;      /* hour              */
     CS_INT      dateminute;    /* minute            */
     CS_INT      datesecond;    /* second            */
     CS_INT      datemsecond;   /* millisecond       */
     CS_INT      datetzone;     /* timezone          */
     CS_INT      datesecfrac;   /* second fractions  */
     CS_INT      datesecprec;   /* precision         */
} CS_DATEREC;
```

where:

- *dateyear* is a value greater than or equal to 1900.
- *datemonth* is a value from 0 to 11.
- *datedmonth* is a value from 1 to 31.
- *datedyear* is a value from 1 to 366.
- *datedweek* is a value from 0 to 6.
- *datehour* is a value from 0 to 23.
- *dateminute* is a value from 0 to 59.
- *datesecond* is a value from 0 to 59.
- *datemsecond* is a value from 0 to 999.
- *datetzone* is reserved for future use. cs_dt_crack does not set this field.
- *datesecfrac* is the number of second fractions. This field is used only with datetime datatypes having a level of precision greater than milliseconds.
- *datesecprec* is the precision. For CS_BIGDATETIME and CS_BIGTIME, this field is always 10^6. This field is used only with datetime datatypes having a level of precision greater than milliseconds.

The meanings of these numbers vary according to an application's locale. For example, if localization information specifies that Sunday is the first day of the week, then a *datedweek* value of 2 represents Tuesday. If localization information specifies that Monday is the first day of the week, then a *datedweek* value of 2 represents Wednesday.

An application can call cs_dt_info to find out what date-related localization values are in effect.

The time zone field (*datetzone*) is reserved for future use. This field will not be set.

See the "International Support" topics page in the *Open Client Client-Library/C Reference Manual*.

Return value

cs_dt_crack returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

The most common reason for a cs_dt_crack failure is an invalid parameter.

Usage

- cs_dt_crack converts a date, time or datetime value into its integer components and places those components into a CS_DATEREC structure.
- Datetime values are stored in an internal format. For example, a CS_DATETIME value is stored as the number of days since January 1, 1900 plus the number of three hundredths of a second since midnight. cs_dt_crack converts a value of this type into a format that an application can more easily access.
- For datetime datatypes with a precision level up to and including milliseconds, second fractions are stored in the *datemsecond* field, and the *datesecfrac* field is not used. For datetime datatypes with a precision level of microseconds and higher, second fractions are stored in the *datesecfrac* field, and the *datemsecond* field is not used. Applications that call the cs_dt_crack routine must therefore determine where to find second fractions based on the datetime datatypes that are being used.

See also

cs_dt_info

cs_dt_info

Description

Sets or retrieves language-specific date, time, or datetime information.

Syntax

```
CS_RETCODE cs_dt_info(context, action, locale, type,
        item, buffer, buflen, outlen)

CS_CONTEXT      *context;
CS_INT          action;
CS_LOCALE       *locale;
CS_INT          type;
CS_INT          item;
CS_VOID         *buffer;
CS_INT          buflen;
CS_INT          *outlen;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

When retrieving date, time, or datetime information, if *locale* is NULL, cs_dt_info uses the default locale information contained in this context structure.

action

One of the following symbolic values:

Value of action	cs_dt_info
CS_SET	Sets a date, time, or datetime conversion format.
CS_GET	Retrieves date, time or datetime information.

locale

A pointer to a CS_LOCALE structure. A locale structure contains locale information, including datetime information.

When setting datetime information, *locale* must be supplied.

When retrieving datetime information, *locale* can be NULL. If *locale* is NULL, cs_dt_info uses the default locale information contained in **context*.

type

The type of information of interest. Table 2-6 lists the symbolic values that are legal for *type*.

item

When retrieving information, *item* is the item number of the type category to retrieve. For example, to retrieve the name of the first month, an application passes *type* as CS_MONTH and *item* as 0.

When setting a datetime conversion format, pass *item* as CS_UNUSED.

buffer
If datetime information is being retrieved, *buffer* points to the space in which cs_dt_info will place the requested information.

If *buflen* indicates that **buffer* is not large enough to hold the requested information, cs_dt_info sets **outlen* to the length of the requested information and returns CS_FAIL.

If a datetime conversion format is being set, *buffer* points to a symbolic value representing a conversion format.

buflen
The length, in bytes, of **buffer*.

If *item* is CS_12HOUR, pass *buflen* as CS_UNUSED.

outlen
A pointer to an integer variable.

cs_dt_info sets **outlen* to the length, in bytes, of the requested information.

If the requested information is larger than *buflen* bytes, an application can use the value of **outlen* to determine how many bytes are needed to hold the information.

Return value

cs_dt_info returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

The most common reason for a cs_dt_info failure is an invalid parameter.

Usage

Table 2-6: Summary of cs_dt_info parameter usage

Value of type	cs_dt_info	action can be	item can be	*buffer is
CS_MONTH	Retrieves the month name string.	CS_GET	0–11	A character string.
CS_SHORTMONTH	Retrieves the short month name string.	CS_GET	0–11	A character string.
CS_DAYNAME	Retrieves the day name string.	CS_GET	0–6	A character string.
CS_DATEORDER	Retrieves the date order string.	CS_GET	CS_UNUSED	A string containing the three characters “m,” “d,” and “y” to indicate the position of the month, day, and year in the datetime format.

Value of type	cs_dt_info	action can be	item can be	*buffer is
CS_12HOUR	Retrieves whether or not the language uses 12-hour time formats.	CS_GET	CS_UNUSED	CS_TRUE if 12-hour formats are used; CS_FALSE if 24-hour formats are used.
CS_DT_CONVFMT	Sets or retrieves the datetime conversion format.	CS_GET or CS_SET	CS_UNUSED	A symbolic value. See the Comments section, below, for a list of possible values.

- cs_dt_info sets or retrieves native language-specific datetime information:
  - cs_dt_info can return native language date part names, date part ordering information, datetime format information, and whether or not the language uses 12-hour date formats.
  - cs_dt_info can set datetime format information.
- If *locale* is NULL, cs_dt_info looks for native language locale information in **context*. An application can set locale information for a CS_CONTEXT by calling cs_config with *property* as CS_LOC_PROP.

  If not specifically set, locale information in a CS_CONTEXT defaults to information that CS-Library picks up from the operating system when the context is allocated. If locale information is not available from the operating system, CS-Library uses platform-specific localization values in the new context.
- A locale's date-order string, which can be retrieved by calling cs_dt_info with *type* as CS_DATEORDER, describes how ambiguous date strings are resolved when converting from character datatypes to CS_DATE, CS_DATETIME or CS_DATETIME4. For example, "04/05/96" could be interpreted as "April 5, 1996" or "May 4, 1996." The former result corresponds to the date-order string of "mdy", and the latter corresponds to "dmy."

  Although an application cannot set a locale's date-order string directly, it can call cs_locale and change the national-language used when converting dates. To do this, the application calls cs_locale with *action* as CS_SET, *type* as CS_LC_TIME, and **buffer* as a locale name. The application can specify a locale whose national language is configured to use a different date-order string. A national language's date-order string is configured as follows:
  - For each national language, a *common.loc* file is located in a language subdirectory in the *$SYBASE/locales/messages* subdirectory.

- The "dateformat" setting in the [datetime] section of the file specifies the date-order string. For example:

```
[datetime]
dateformat=dmy
```

See the *Open Client and Server Configuration Guide for Microsoft Windows* or *Open Client and Server Configuration Guide for UNIX*.

- The date conversion format, which can be set or retrieved by calling cs_dt_info with *type* as CS_DT_CONVFMT, describes the format of the result when a CS_DATE, CS_TIME, CS_DATETIME, CS_DATETIME4, CS_BIGDATETIME or CS_BIGTIME value is converted to a character-based datatype.
- Table 2-7 lists the values that are legal for **buffer* when *type* is CS_DT_CONVFMT in conversions from CS_CHAR to CS_DATETIME, CS_DATE, or CS_TIME. This conversion format is also used to describe results when a character string is converted to any of these datetime datatypes.

Table 2-7: Values for *buffer when type is CS_DT_CONVFMT (cs_dt_info)

Symbolic value	CS_CHAR converted from CS_DATETIME, for example: Aug 24 2009 5:36PM	CS_CHAR converted from CS_DATE, for example: Aug 24 2009	CS_CHAR converted from CS_TIME, for example: 5:36PM
CS_DATES_HM	hh:mm 17:36	hh:mm 00:00	hh:mm 17:36
CS_DATES_HMA	hh:mm[AM|PM] 5:36PM	hh:mm 12:00AM	hh:mm 5:36PM
CS_DATES_HMS	hh:mm:ss 17:36:00	hh:mm:ss 00:00:00	hh:mm:ss 17:36:00
CS_DATES_HMS_ALT	hh:mm:ss 17:36:32	hh:mm:ss 00:00:00	hh:mm:ss 17:36:32
CS_DATES_HMSZA	hh:mm:ss:zzz[AM|PM] 5:36:00:000PM	hh:mm:ss:zzz[AM|PM] 12:00:00:000AM	hh:mm:ss:zzz[AM|PM] 5:36:00:000PM
CS_DATES_HMSZ	hh:mm:ss:zzz 17:36:00:000	hh:mm:ss:zzz 00:00:00:000	hh:mm:ss:zzz 17:36:00:000
CS_DATES_LONG	mon dd yyyy hh:mm:ss:zzz [AM|PM] Aug 24 2009 05:36:00:000PM	mon dd yyyy Aug 24 2009	hh:mm:ss:zzz [AM|PM] 05:36:00:000PM

Symbolic value	CS_CHAR converted from CS_DATETIME, for example: Aug 24 2009 5:36PM	CS_CHAR converted from CS_DATE, for example: Aug 24 2009	CS_CHAR converted from CS_TIME, for example: 5:36PM
CS_DATES_LONG_ALT	mon dd yyyy hh:mm:ss:zzz [AM|PM] `Aug 24 2009 05:36:00:000PM`	mon dd yyyy hh:mm:ss:zzz [AM|PM] `Aug 24 2009 12:00:00:000 AM`	mon dd yyyy hh:mm:ss:zzz [AM|PM] `Jan 01 1900 05:36:00:000 PM`
CS_DATES_MDYHMS	mon dd yyyy hh:mm:ss `Aug 24 2009 17:36:00`	mon dd yyyy `Aug 24 2009`	hh:mm:ss `17:36:00`
CS_DATES_MDYHMS_ALT	mon dd yyyy hh:mm:ss `Aug 24 2009 17:36:00`	mon dd yyyy hh:mm:ss `Aug 24 2009 00:00:00`	mon dd yyyy hh:mm:ss `Jan 1 1900 17:36:00`
CS_DATES_SHORT	mon dd yyyy hh:mm [AM|PM] `Aug 24 2009 5:36PM`	mon dd yyyy `Aug 24 2009`	hh:mm [AM|PM] `5:36PM`
CS_DATES_SHORT_ALT	mon dd yyyy hh:mm [AM|PM] `Aug 24 2009 5:36PM`	mon dd yyyy hh:mm [AM|PM] `Aug 24 2009 12:00AM`	mon dd yyyy hh:mm [AM|PM] `Jan 1 1900 5:36PM`
CS_DATES_DMY1	dd/mm/yy `24/08/09`	dd/mm/yy `24/08/09`	
CS_DATES_DMY1_YYYY	dd/mm/yyyy `24/08/2009`	dd/mm/yyyy `24/08/2009`	
CS_DATES_DYM1	dd/yy/mm `24/09/08`	dd/yy/mm `24/09/08`	
CS_DATES_DYM1_YYYY	dd/yyyy/mm `24/2009/08`	dd/yy/mm `24/2009/08`	
CS_DATES_MDY1	mm/dd/yy `08/24/09`	mm/dd/yy `08/24/09`	
CS_DATES_MDY1_YYYY	mm/dd/yyyy `08/24/2009`	mm/dd/yyyy `08/24/2009`	
CS_DATES_MYD1	mm/yy/dd `08/09/24`	mm/yy/dd `08/2009/24`	
CS_DATES_MYD1_YYYY	mm/yyyy/dd `08/2009/24`	mm/yyyy/dd `08/2009/24`	
CS_DATES_YDM1	yy/dd/mm `09/24/08`	yy/dd/mm `09/24/08`	

Symbolic value	CS_CHAR converted from CS_DATETIME, for example: Aug 24 2009 5:36PM	CS_CHAR converted from CS_DATE, for example: Aug 24 2009	CS_CHAR converted from CS_TIME, for example: 5:36PM
CS_DATES_YDM1_YYYY	yyyy/dd/mm 2009/24/08	yyyy/dd/mm 2009/24/08	
CS_DATES_YMD1	yy.mm.dd 09.08.24	yy.mm.dd 09.08.24	
CS_DATES_YMD1_YYYY	yyyy.mm.dd 2009.08.24	yyyy.mm.dd 2009.08.24	
CS_DATES_DMY2	dd.mm.yy 24.08.09	dd.mm.yy 24.08.09	
CS_DATES_DMY2_YYYY	dd.mm.yyyy 24.08.2009	dd.mm.yyyy 24.08.2009	
CS_DATES_MDY2	mon dd, yy Aug 24,09	mon dd, yy Aug 24,09	
CS_DATES_MDY2_YYYY	mon dd, yyyy Aug 24,2009	mon dd, yyyy Aug 24,2009	
CS_DATES_YMD2	yy/mm/dd 09/08/24	yy/mm/dd 09/08/24	
CS_DATES_YMD2_YYYY	yyyy/mm/dd 2009/08/24	yyyy/mm/dd 2009/08/24	
CS_DATES_DMY3	dd-mm-yy 24-08-09	dd-mm-yy 24-08-09	
CS_DATES_DMY3_YYYY	dd-mm-yyyy 24-08-2009	dd-mm-yyyy 24-08-2009	
CS_DATES_MDY3	mm-dd-yy 08-24-09	mm-dd-yy 08-24-09	
CS_DATES_MDY3_YYYY	mm-dd-yyyy 08-24-2009	mm-dd-yyyy 08-24-2009	
CS_DATES_YMD3	yymmdd 090824	yymmdd 090824	
CS_DATES_YMD3_YYYY	yyyymmdd 20090824	yyyymmdd 20090824	
CS_DATES_YMDTHMS 23	yyyy-mm-ddThh:mm:ss 2009-08-24T17:36:00	yyyy-mm-dd 2009-08-24	hh:mm:ss 17:36:00
CS_DATES_DMY4	dd mon yy 24 Aug 09	dd mon yy 24 Aug 09	

Symbolic value	CS_CHAR converted from CS_DATETIME, for example: Aug 24 2009 5:36PM	CS_CHAR converted from CS_DATE, for example: Aug 24 2009	CS_CHAR converted from CS_TIME, for example: 5:36PM
CS_DATES_DMY4_YYYY	dd mon yyyy `24 Aug 2009`	dd mon yyyy `24 Aug 2009`	

- Table 2-8 lists the values that are legal for **buffer* when *type* is CS_DT_CONVFMT in conversions between CS_CHAR and CS_BIGDATETIME and CS_BIGTIME:

Table 2-8: Values for *buffer when type is CS_DT_CONVFMT (cs_dt_info)

Symbolic value	CS_CHAR converted from CS_BIGDATETIME, for example: Aug 24 2009 5:36PM	CS_CHAR converted from CS_BIGTIME, for example: 5:36PM
CS_DATES_ HMSUSA, or CS_DATES_ HMSUSA_YYYY	hh:mm:ss.zzzzzz[AM|PM] `5:36:00.000000PM`	hh:mm:ss.zzzz[AM|PM] `5:36:00.000000PM`
CS_DATES_ HMSUS, or CS_DATES_ HMSUS_YYYY	hh:mm:ss.zzzzzz `17:36:00.000000`	hh:mm:ss.zzzzzz `17:36:00.000000`
CS_DATES_ LONGUSA	mon dd yy hh:mm:ss.zzzzzz[AM|PM] `Aug 24 09 5:36:00.000000PM`	mon dd yy hh:mm:ss.zzzzzz[AM|PM] `Jan 1 01 5:36:00.000000PM`
CS_DATES_ LONGUSA_YYYY	mon dd yyyy hh:mm:ss.zzzzzz[AM|PM] `Aug 24 2009 5:36:00.000000PM`	mon dd yyyy hh:mm:ss.zzzzzz[AM|PM] `Jan 1 0001 5:36:00.000000PM`
CS_DATES_ LONGUS	mon dd yy hh:mm:ss.zzzzzz `Aug 24 09 17:36:00.000000`	mon dd yy hh:mm:ss.zzzzzz `Jan 1 01 17:36:00.000000`
CS_DATES_ LONGUS_YYYY	mon dd yyyy hh:mm:ss.zzzzzz `Aug 24 2009 17:36:00.000000`	mon dd yy hh:mm:ss.zzzzzz `Jan 0 0001 17:36:00.000000`
CS_DATES_ YMDHMSUS_YYYY	yyyy-mm-dd hh:mm:ss.zzzzzz `2009-08-24 17:36:00.000000`	yyyy-mm-dd hh:mm:ss.zzzzzz `0001-01-01 17:36:00.000000`

- A cs_locale (CS_SET, CS_LC_TIME) call or a cs_locale (CS_SET, CS_LC_ALL) call resets date/time conversion information to the default settings for the specified national language.

See also cs_dt_crack, cs_locale

cs_loc_alloc

Description

Allocates a CS_LOCALE structure.

Syntax

CS_RETCODE cs_loc_alloc(context, loc_pointer)

```
CS_CONTENT      *context;
CS_LOCALE       **loc_pointer;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

loc_pointer

The address of a pointer variable. cs_loc_alloc sets **loc_pointer* to the address of a newly allocated CS_LOCALE structure.

Return value

cs_loc_alloc returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

The most common reason for a cs_loc_alloc failure is a lack of adequate memory.

Usage

- An Open Client and Open Server application can use a CS_LOCALE structure to define custom localization values for a context, thread, connection, or data element. To define custom localization values, an application:
  - Calls cs_loc_alloc to allocate a CS_LOCALE structure.
  - Calls cs_locale (CS_SET) to load the CS_LOCALE with custom values.
  - Uses the CS_LOCALE to set the CS_LOC_PROP property for a context or connection; calls srv_thread_props to set the SRV_T_LOCALE property for a thread; uses the CS_LOCALE in a CS_DATAFMT structure that describes a program variable; or uses the CS_LOCALE as a parameter to an Open Client and Open Server routine.
  - Calls cs_loc_drop to drop the CS_LOCALE.
- Localization values define:
  - The language and character set to use for Open Client and Open Server and Adaptive Server Enterprise messages
  - How to represent dates and times

- The character set to use when converting data to and from character datatypes
- The collating sequence used to define the sort order used by cs_strcmp

See also cs_ctx_alloc, cs_loc_drop, cs_locale

cs_loc_drop

Description Deallocates a CS_LOCALE structure.

Syntax CS_RETCODE cs_loc_drop(context, locale)

```
CS_CONTEXT      *context;
CS_LOCALE       *locale;
```

Parameters

context
A pointer to the CS_CONTEXT structure that represents the context in which the CS_LOCALE was allocated.

locale
A pointer to a CS_LOCALE structure.

Return value cs_loc_drop returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- A CS_LOCALE structure contains localization information.
- Once a CS_LOCALE structure has been deallocated, it cannot be used again. To allocate a new CS_LOCALE structure, an application can call cs_loc_alloc.
- An application should take care to ensure that it does not deallocate a CS_LOCALE structure that is still in use. A CS_LOCALE structure is considered to be in use if a CS_DATAFMT structure references it.
- An application can deallocate a CS_LOCALE structure after calling cs_config or ct_con_props to set the CS_LOC_PROP property for a context or connection. This is because cs_config and ct_con_props copy information from the user-supplied CS_LOCALE structure rather than setting up direct references to it.

See also cs_loc_alloc, cs_locale

cs_locale

Description Loads a CS_LOCALE structure with localization values or retrieve the locale name previously used to load a CS_LOCALE structure.

Syntax

```
CS_RETCODE cs_locale(context, action, locale, type,
        buffer, buflen, outlen)

CS_CONTEXT      *context;
CS_INT          action;
CS_LOCALE       *locale;
CS_INT          type;
CS_CHAR         *buffer;
CS_INT          buflen;
CS_INT          *outlen;
```

Parameters

context

A pointer to the CS_CONTEXT structure that represents the context in which the CS_LOCALE was allocated.

action

One of the following symbolic values:

Value of action	cs_locale
CS_SET	Loads the CS_LOCALE with new localization values.
CS_GET	Retrieves the locale name that was used to load the CS_LOCALE.

locale

A pointer to a CS_LOCALE structure. If *action* is CS_SET, cs_locale modifies this structure. If action is CS_GET, cs_locale examines the structure to determine the locale name that was previously used to load it.

type

One of the following symbolic values:

Value of type	Indicates
CS_LC_ALL	All types of localization information. **Note** CS_LC_ALL is "set only"; that is, *action* must be CS_SET when *type* is CS_LC_ALL.
CS_LC_COLLATE	The collating sequence (also called "sort order"). Open Client uses a collating sequence when sorting and comparing character data.
CS_LC_CTYPE	The character set. Open Client uses a character set when it converts to or from character datatypes.
CS_LC_MESSAGE	The language and character set to use for Open Client and Open Server and Adaptive Server Enterprise error messages.
CS_LC_TIME	The language and character set to use when converting between datetime and character datatypes. CS_LC_TIME controls month names and abbreviations, datepart ordering, and whether the "am/pm" string is used.
CS_SYB_LANG, CS_SYB_CHARSET, CS_SYB_SORTORDER, CS_SYB_LANG_CHARSET	For information on these values, see "Using language, character set, and sort order names with cs_locale" on page 66.

Warning! Open Server application programmers must set type to CS_LC_ALL when configuring the CS_LOCALE structure that applies to the Open Server application as a whole.

buffer

If action is CS_SET, *buffer* points to a character string that represents a locale name, a character set name, a language name, a sort order name, or a language/character set pair.

If action is CS_GET, *buffer* points to the space in which cs_locale will place a locale name, a character set name, a language name, a sort order name, or a language/character set pair. On output, all names are null-terminated. The buffer must be long enough for the name plus a null terminator.

buflen
The length, in bytes, of **buffer*.

If *action* is CS_SET and the value in **buffer* is null-terminated, pass *buflen* as CS_NULLTERM.

outlen
A pointer to an integer variable.

outlen is not used if *action* is CS_SET.

If *action* is CS_GET and *outlen* is supplied, cs_locale sets **outlen* to the length, in bytes, of the locale name.

If the name is larger than *buflen* bytes, an application can use the value of **outlen* to determine how many bytes are needed to hold the name.

If *action* is CS_SET or if an application does not require return length information, it can pass *outlen* as NULL.

Return value

cs_locale returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Common reasons for a cs_locale failure include:

- *action* is CS_SET and the **buffer* locale name cannot be found in the Sybase locales file.
- *action* is CS_GET and *buflen* indicates that the **buffer* data space is too small.
- Missing localization files.

Usage

Note cs_locale's behavior depends on platform-specific configuration issues. You must read the localization chapter in the *Open Client and Open Server Configuration Guide for Microsoft Windows* and *Open Client and Open Server Configuration Guide for UNIX* to obtain a full understanding of Client-Library's localization mechanism. For a discussion of programming issues related to localization, see the *Open Client and Open Server International Developers Guide*.

- cs_locale(CS_SET) loads a CS_LOCALE structure with localization values. cs_locale(CS_GET) retrieves current settings from the CS_LOCALE structure.

- A *locale name* is a character string that represents a language/ character set/sort order combination. For example, the locale name "fr" might represent the language/character set/sort order combination "French, iso_1, binary."
  - Sybase predefines some locale names in the default locales file.
  - A System Administrator can define additional locale names and add them to the Sybase locales file. The *Open Client and Open Server Configuration Guide for Microsoft Windows* and *Open Client and Open Server Configuration Guide for UNIX* contains instructions for adding locale names.
- See the *Open Client and Open Server International Developers Guide*.

Loading a CS_LOCALE structure

- An application needs to initialize, or "load," a CS_LOCALE before using it to define custom localization values for a context, connection, or data element.
- cs_locale(CS_SET) loads a CS_LOCALE structure with localization values. Any localization value can be specified by giving a locale name. Character sets, languages, and sort orders can also be specified directly by name.
- When specifying a locale name, *buffer* must specify a name that corresponds to an entry in the Sybase locales file.

  buffer can also be passed as NULL to specify the default locale. In this case, cs_locale searches the operating system for a locale name to use. If an appropriate locale name cannot be found in the operating system environment, cs_locale uses a platform-dependent default locale name.

  The localization item(s) of interest are loaded based on the configuration of the locales file entry. See the *Open Client and Open Server Configuration Guide* for your platform.
- For instructions for directly specifying character set, language, or sort order names, see "Using language, character set, and sort order names with cs_locale" on page 66.
- After loading a CS_LOCALE with custom values, an application can:
  - Call cs_config with *property* as CS_LOC_PROP to copy the custom localization values into a context structure.
  - Call ct_con_props with *property* as CS_LOC_PROP to copy the custom localization values into a connection structure.

- Supply the CS_LOCALE as a parameter to a routine that accepts custom localization values (cs_dt_info, cs_strcmp, cs_time).
- Include the CS_LOCALE in a CS_DATAFMT structure describing a destination program variable (cs_convert, ct_bind).

- Because cs_config copies locale information, an application can deallocate a CS_LOCALE structure after calling cs_config to set the CS_LOC_PROP property. Likewise, an application can deallocate a CS_LOCALE structure after calling ct_con_props to set the CS_LOC_PROP property. If a CS_DATAFMT structure uses a CS_LOCALE structure, however, the application must not deallocate the CS_LOCALE until the CS_DATAFMT no longer references it.
- The first time a locale name is referenced, all localization information for the language, character set, and sort order that the locale name identifies is read from the environment and cached into **context*. If this locale name is referenced again, cs_locale reads the information from the CS_CONTEXT instead of the environment.

Retrieving a locale name

- An application can retrieve the locale name that was used to load a CS_LOCALE by calling cs_locale(CS_GET) with *type* as the type of localization information of interest and *locale* as a pointer to the CS_LOCALE structure.
- cs_locale sets **buffer* to a null-terminated character string representing the locale name that was used to load the CS_LOCALE.

Using language, character set, and sort order names with *cs_locale*

- It is possible for an application to use language, character set, and sort order names, instead of a locale name, when calling cs_locale.
- To use a language, character set, or sort order name, an application calls cs_locale with *type* as CS_SYB_LANG, CS_SYB_CHARSET, CS_SYB_SORTORDER, or CS_SYB_LANG_CHARSET. The following table summarizes cs_locale parameters for these values of *type*:

Table 2-9: Using language, character set, and sort order names with cs_locale

Value of type	action is	buffer is	cs_locale
CS_SYB_LANG	CS_SET	A pointer to a language name.	Loads the CS_LOCALE with the specified language information.
	CS_GET	A pointer to data space.	Places the current language name in **buffer*. The name is null terminated.

Value of type	action is	buffer is	cs_locale
CS_SYB_CHARSET	CS_SET	A pointer to a character set name.	Loads the CS_LOCALE with the specified character set information.
	CS_GET	A pointer to data space.	Places the current character set name in **buffer*. The name is null terminated.
CS_SYB_SORTORDER	CS_SET	A pointer to a sort order name.	Loads the CS_LOCALE with the specified sort order information.
	CS_GET	A pointer to data space.	Places the current sort order name in **buffer*. The name is null terminated.
CS_SYB_LANG_CHARSET	CS_SET	A pointer to a string of the form *language_name*. *character_set_name*.	Loads the CS_LOCALE with the specified language and character set information.
	CS_GET	A pointer to data space.	Places a string of the form *language_name.character_set_ name* in **buffer*. The string is null terminated.

- The application must have previously loaded the CS_LOCALE structure with consistent information by calling cs_locale with *type* as CS_LC_ALL.
- If an application specifies only a language name, then cs_locale uses the character set and sort order already specified in the preloaded CS_LOCALE structure.

  If an application specifies only a character-set name, then cs_locale uses the language and sort order already specified in the preloaded CS_LOCALE structure.

  If an application specifies only a sort-order name, then cs_locale uses the language and character set already specified in the preloaded CS_LOCALE structure.

  If a language, character set, and sort-order combination is not valid, cs_locale returns CS_FAIL.
- Valid language names correspond to subdirectories in the *$SYBASE/locales* directory. Valid character-set names correspond to subdirectories in the *$SYBASE/charsets* directory. Valid sort-order names for a character set correspond to file names, stripped of any suffix, in the *$SYBASE/charsets/character_set_name* directory.
- If the required localization files for the requested language or character set do not exist, cs_locale returns CS_FAIL.

See also cs_loc_alloc, cs_loc_drop

cs_manage_convert

Description

Installs or retrieves a user-defined character-set conversion routine.

Syntax

```
CS_RETCODE cs_manage_convert(context, action,
      srctype, srcname, srcnamelen,
      desttype, destname, destnamelen,
      conv_multiplier, func)

CS_CONTEXT      *context;
CS_INT          action;
CS_INT          srctype;
CS_CHAR         *srcname;
CS_INT          srcnamelen;
CS_INT          desttype;
CS_CHAR         *destname;
CS_INT          destnamelen;
CS_INT          *conv_multiplier;
CS_CONV_FUNC    *func;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

action

One of the following symbolic values:

Value of action	cs_manage_convert
CS_SET	Installs a conversion routine and conversion multiplier for conversions between the indicated datatypes and character-set names.
CS_GET	Retrieves the current conversion routine and conversion multiplier for the indicated datatypes and character-set names.
CS_CLEAR	Clears the current conversion routine by replacing it with CS-Library's default conversion routine for the indicated datatypes and character-set names.

srctype

The datatype of the source data for the conversion. In the current version, *srctype* must be CS_CHAR_TYPE.

srcname

The name of the character set associated with *srctype*. This name must correspond to the name of a subdirectory within the *charsets* subdirectory of the Sybase installation directory.

srcnamelen

The length, in bytes, of *srcname*. If *srcname* is null-terminated, *srcnamelen* can be passed as CS_NULLTERM.

desttype
The datatype of the destination data. In the current version, *desttype* must be CS_CHAR_TYPE.

destname
The name of the destination character set. This name must correspond to the name of a subdirectory within the *charsets* subdirectory of the Sybase installation directory.

destnamelen
The length, in bytes, of *destname*. If *destname* is null-terminated, *destnamelen* can be passed as CS_NULLTERM.

conv_multiplier
The address of a CS_INT variable. When action is CS_SET, pass **conv_multiplier* as the conversion multiplier for the indicated character-set conversion. When action is CS_GET, **conv_multiplier* receives the conversion multiplier for the indicated character-set conversion. When action is CS_CLEAR, pass *conv_multiplier* as NULL.

See "Meaning of the conversion multiplier" on page 71 for a explanation of how applications use this number.

func
The address of a CS_CONV_FUNC variable, which itself is a pointer to a character-set conversion routine. "Defining a custom character set conversion routine" on page 71 describes the requirements for coding a custom character-set conversion routine.

If a conversion routine is being installed, **func* points to the conversion routine to be installed.

If a conversion routine is being retrieved, cs_manage_convert sets **func* to point to the currently installed character-set conversion routine for *srcname* to *destname* conversions, or to NULL if no custom routine is installed.

If a conversion routine is being cleared, pass **func* as NULL.

Note *func* represents a pointer to a pointer to a function. There are special requirements for passing this parameter. See the example code fragment under "Installing a custom character set conversion routine" on page 73.

Return value

cs_manage_convert returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.

Returns	Indicates
CS_FAIL	The routine failed.

The most common reason for a cs_manage_convert failure is an invalid parameter.

Usage

- cs_manage_convert allows an application to install a custom character-set conversion routine that converts data from one character set to another.

Character set conversion

- Client-Library, CS-Library, and Server-Library can all perform character-set conversion. Character-set conversion occurs when an application converts between any two character datatypes and associates different character sets with the source and destination.
  - In CS-Library, cs_convert performs character-set conversion when converting between two character datatypes if the *destfmt* CS_DATAFMT structure specifies (or defaults to) a different locale than the *srcfmt* CS_DATAFMT structure.
  - In Client-Library, an application can request character-set conversion for fetched character data by binding the column to a character-datatype variable and passing a pointer to a CS_LOCALE in ct_bind's *datafmt* that is different from the connection's locale (that is, the CS_LOC_PROP connection property).
  - In Server-Library, all character data sent to a client or received from a client is automatically converted between the client thread's character set and the Open Server character set.
- The character datatypes are CS_CHAR, CS_LONGCHAR, CS_TEXT, CS_UNITEXT, CS_UNICHAR, CS_VARCHAR and CS_XML.
- cs_manage_convert requires an application to pass both *srctype* and *desttype* as CS_CHAR_TYPE. However, CS-Library, Client-Library, and Server-Library will call the conversion routine to convert between any two character-based types when the conversion locales specify the character sets associated with the conversion routine.
- The most common reason for installing a custom conversion routine is to improve performance by replacing an indirect conversion with a direct conversion.

A custom character-set conversion routine can improve performance in applications that rely on character-set conversions where CS-Library does not use direct character-set conversion. Indirect character-set conversion converts first to Unicode UTF-8, and then from Unicode UTF-8 to the destination character set. Applications that perform these conversions can improve performance by installing a custom routine that supports direct conversion.

For example, an Open Server application could install a custom routine to convert between ISO 8859-1 and EUC JIS. This direct conversion may be faster than the indirect conversion (ISO 8859-1 to/from Unicode UTF-8 to/from EUC JIS) that is supplied with Open Server.

- To find out whether a specific character conversion is direct or indirect, look in the source character set's conversion configuration file. If there is an entry for the destination character set, then the conversion is direct. Character set configuration files are described in the *Open Client and Open Server International Developers Guide*.
- See the *Open Client and Open Server International Developers Guide*.

Meaning of the conversion multiplier

- Applications must provide cs_manage_convert with a conversion multiplier for conversions between the indicated character sets.
- The value of the conversion multiplier equals the largest number of bytes in the destination result that can replace one source byte when converting between the indicated character sets.
- Applications can retrieve the conversion multiplier for a specific character-set conversion with cs_conv_mult. This number allows the application to determine the destination space needed for a conversion.

Defining a custom character set conversion routine

- A custom character-set conversion routine is defined as follows:

```
CS_RETCODE CS_PUBLIC
 convfunc(context, srcfmt, srcdata,
          destfmt, destdata, destlen)
 CS_CONTEXT          *context;
 CS_DATAFMT          *srcfmt;
 CS_VOID             *srcdata;
 CS_DATAFMT          *destfmt;
 CS_VOID             *destdata;
 CS_INT              *destlen;
```

where:

- *context* is a pointer to a CS_CONTEXT structure.
- *srcfmt* is a pointer to a CS_DATAFMT structure describing the source data. *srcfmt–>maxlength* describes the actual length, in bytes, of the source data.
- *srcdata* is a pointer to the source data.
- *destfmt* is a pointer to a CS_DATAFMT structure describing the destination data. *destfmt–>maxlength* describes the actual length, in bytes, of the destination data space.
- *destdata* is a pointer to the destination data space.

destlen is a pointer to an integer. The conversion routine should set **destlen* to the number of bytes placed in **destdata*. If the routine writes a truncated result, it should set **destlen* as the number of bytes written before truncation.

Note When converting into a CS_VARCHAR structure, the conversion routine should set both **destlen* and the CS_VARCHAR's *len* field to the number of bytes written to the CS_VARCHAR's *str* field.

- cs_config is the only CS-Library, Client-Library, or Server-Library function that can be called from within a custom conversion routine.
- A custom character-set conversion routine can return any of the values listed in Table 2-10.
  - If the conversion routine returns a value from Table 2-10 other than CS_SUCCEED, then the application receives a Client-Library or CS-Library message that corresponds to the indicated error condition.
  - If the conversion routine returns a value that is not listed in Table 2-10, then the application receives an "Unknown return code" error message from Client-Library or CS-Library.

Table 2-10: Return values for a custom conversion routine

Return value	Indicates
CS_SUCCEED	Successful conversion.
CS_TRUNCATED	The conversion resulted in truncation.
CS_MEM_ERROR	A memory allocation failure has occurred.
CS_EBADXLT	Some characters could not be translated.
CS_ENOXLT	The requested translation is not supported.
CS_EDOMAIN	The source value is outside the domain of legal values for the datatype.
CS_EDIVZERO	Division by 0 is not allowed.
CS_EOVERFLOW	The conversion resulted in overflow.
CS_EUNDERFLOW	The conversion resulted in underflow.
CS_EPRECISION	The conversion resulted in loss of precision.
CS_ESCALE	An illegal scale value was encountered.
CS_ESYNTAX	The conversion resulted in a value which is not syntactically correct for the destination type.
CS_ESTYLE	The conversion operation was stopped due to a style error.

Installing a custom character set conversion routine

- The following code demonstrates calling cs_manage_convert to install a custom conversion routine. The code is based on the assumption that the installed routine has been defined correctly. (See “Defining a custom character set conversion routine” on page 71.) The program variable *p_conv_func* is used to pass the address of the conversion routine.

```
#define MULT_ISO_1_TO_EUCJIS 4
 CS_CONV_FUNC p_conv_func;
 CS_INT       conv_mult = MULT_ISO_1_TO_EUCJIS;

/*
 ** Install the routine charconv_iso_1_TO_eucjis() to convert
 ** character data from iso_1 character set to eucjis character
 ** set.
 */
 p_conv_func = charconv_iso_1_TO_eucjis;
 if (cs_manage_convert(context, CS_SET,
         CS_CHAR_TYPE, "iso_1", CS_NULLTERM,
         CS_CHAR_TYPE, "eucjis", CS_NULLTERM,
         &conv_mult, &p_conv_func )
     != CS_SUCCEED)
 {
```

```
        fprintf(stdout, "cs_manage_convert() failed!\n");
        (CS_VOID)ct_exit(context, CS_FORCE_EXIT);
        (CS_VOID)cs_ctx_drop(context);
        exit(-1);
    }
```

See also cs_conv_mult, cs_convert, cs_locale, cs_set_convert

cs_objects

Description Saves, retrieves, or clears objects and data associated with them.

Syntax

```
CS_RETCODE cs_objects(context, action, objname,
        objdata)

CS_CONTEXT      *context;
CS_INT          action;
CS_OBJNAME      *objname;
CS_OBJDATA      *objdata;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

action

One of the following symbolic values:

Value of action	cs_objects
CS_SET	Saves an object.
CS_GET	Retrieves the first matching object that it finds.
CS_CLEAR	Clears all matching objects.

objname

A pointer to an object name structure. **objname* names and describes the object of interest. An object name structure is defined as follows:

```
/*
** CS_OBJNAME
*/
typedef struct _cs_objname
{
    CS_BOOL         thinkexists;
    CS_INT          object_type;
    CS_CHAR         last_name[CS_MAX_CHAR];
    CS_INT          lnlen;
    CS_CHAR         first_name[CS_MAX_CHAR];
```

```
    CS_INT          fnlen;
    CS_VOID         *scope;
    CS_INT          scopelen;
    CS_VOID         *thread;
    CS_INT          threadlen;
  } CS_OBJNAME;
```

The *object_type*, *last_name*, *first_name*, *scope*, and *thread* fields form a five-part key that identifies a stored object (see “cs_objects naming keys” on page 79). Table 2-11 describes the CS_OBJNAME fields:

Table 2-11: CS_OBJNAME fields

Field	Description	Notes
thinkexists	Indicates whether the application expects this object to exist.	The value of *thinkexists* affects the cs_objects return code. For more information, see the Return values.
object_type	The type of the object.	This field is the first part of a five-part key. *object_type* can be one of these values: • CS_CONNECTNAME • CS_CURSORNAME • CS_STATEMENTNAME • CS_CURRENT_CONNECTION • CS_WILDCARD (matches any value) • A user-defined value. User-defined values must be >= 100.
last_name	The "last name" associated with the object of interest, if any.	This field is the second part of a five-part key.
lnlen	The length, in bytes, of *last_name*.	Can be CS_NULLTERM to indicate a null-terminated *last_name*. Can be CS_UNUSED to indicate an internal "unused" value for *last_name*. For CS_GET and CS_CLEAR operations, can be CS_WILDCARD to match any *last_name* value.
first_name	The "first name" associated with the object of interest, if any.	This field is the third part of a five-part key.
fnlen	The length, in bytes, of *first_name*.	Can be CS_NULLTERM to indicate a null-terminated *first_name*. Can be CS_UNUSED to indicate an internal "unused" value for *first_name*. For CS_GET and CS_CLEAR operations, can be CS_WILDCARD to match any *first_name* value.
scope	Data that describes the scope of the object.	This field is the fourth part of a five-part key.

Field	Description	Notes
scopelen	The length, in bytes, of *scope*.	Can be CS_NULLTERM to indicate null-terminated scope data. Can be CS_UNUSED to indicate an internal "unused" value for **scope*. For CS_GET and CS_CLEAR operations, can be CS_WILDCARD to match any *scope* value.
thread	Platform-specific data that is used to distinguish threads in a multi-threaded execution environment.	This field is the fifth part of a five-part key.
threadlen	The length, in bytes, of *thread*.	Can be CS_NULLTERM to indicate null-terminated thread data. Can be CS_UNUSED to indicate an internal "unused" value for **thread*. For CS_GET and CS_CLEAR operations, can be CS_WILDCARD to match any *thread* value.

objdata

A pointer to an object data structure. **objdata* is the object of interest and any data associated with it. An object data structure is defined as follows:

```
/*
 ** CS_OBJDATA
 */
 typedef struct _cs_objdata
 {
   CS_BOOL           actuallyexists;
   CS_CONNECTION     *connection;
   CS_COMMAND        *command;
   CS_VOID           *buffer;
   CS_INT            buflen;
 } CS_NAMEDATA;
```

Table 2-12 describes the CS_OBJDATA fields:

Table 2-12: CS_OBJDATA fields

Field	Description	Notes
actuallyexists	Indicates whether this object actually exists.	cs_objects sets *actuallyexists* to CS_TRUE if it finds a matching object. cs_objects sets *actuallyexists* to CS_FALSE if it does not find a matching object.
connection	A pointer to the CS_CONNECTION structure representing the connection in which the object exists.	
command	A pointer to the CS_COMMAND structure representing the command space with which the object is associated.	Can be NULL.
buffer	A pointer to data space. An application can use *buffer* to associate data with a saved object.	If *action* is CS_SET, **buffer* contains the data to associate with the object. If *action* is CS_GET, cs_objects sets **buffer* to the data associated with the object being retrieved.
buflen	The length, in bytes, of **buffer*.	If *action* is CS_SET, *buflen* is the length of the data contained in **buffer*. Can be CS_NULLTERM to indicate null-terminated data. Can be CS_UNUSED to indicate that there is no data associated with the object being saved. If *action* is CS_GET, *buflen* is the maximum capacity of **buffer*. cs_objects overwrites *buflen* with the number of bytes copied to **buffer*. If *buflen* is CS_UNUSED, cs_objects overwrites *buflen* with the length of the data but does not copy it to **buffer*.

Return value

cs_objects returns CS_SUCCEED or CS_FAIL depending on the values passed as *action* and *objname–>thinkexists* (See Table 2-11 on page 76). Table 2-13 lists the return code for each combination:

Table 2-13: cs_objects return values

cs_objects Called with		cs_objects returns		
action As	objname→thinkexists As	No match	Last-name match	Full match
CS_GET	CS_TRUE	CS_FAIL	CS_FAIL	CS_SUCCEED
CS_GET	CS_FALSE	CS_SUCCEED	CS_SUCCEED	CS_SUCCEED
CS_SET	CS_TRUE	CS_FAIL	CS_FAIL	CS_SUCCEED
CS_SET	CS_FALSE	CS_SUCCEED	CS_SUCCEED	CS_FAIL
CS_CLEAR	CS_TRUE	CS_FAIL	CS_FAIL	CS_SUCCEED
CS_CLEAR	CS_FALSE	CS_SUCCEED	CS_SUCCEED	CS_SUCCEED

Usage

Table 2-14: Summary of cs_objects parameter usage

Value of action	objname is	objdata is
CS_SET	A five-part key for the object.	The object to save and any additional data to save with it.
CS_GET	A five-part key for the object.	Set to the retrieved object.
CS_CLEAR	A five-part key for the object.	CS_UNUSED.

- cs_objects is useful in precompiler applications that need to retrieve structures and data items by name.

cs_objects naming keys

- cs_objects uses a five-part key, composed of the *object_type*, *last_name*, *first_name*, *scope*, and *thread* fields of **objname* structure.
  - On CS_SET operations, cs_objects uses this key to store the **objdata* object.
  - On CS_GET operations, cs_objects uses this key to retrieve an object specification into **objdata*.
  - On CS_CLEAR operations, cs_objects clears all objects that match the key.
- Table 2-15 describes the rules that cs_objects uses to determine whether or not key fields match:

Table 2-15: cs_objects key matching rules

*objname key length is	Stored key length is CS_UNUSED	Stored key length is another legal value
CS_WILDCARD	Match	Match
CS_UNUSED	Match	No match
Another Legal Value	No match	Match, if the names match and have the same length.

- cs_objects can achieve two types of matches:
  - "last-name matches," in which the *last_name*, *scope*, and *thread* parts of the key match.
  - "full matches," in which all five parts of the key match.

  The type of match that cs_objects achieves, together with *action* and *objname–>thinkexists*, determine its return code.
- On CS_GET and CS_CLEAR operations, an application may specify CS_WILDCARD for one or more **objname* key fields:
  - On a CS_GET operation, cs_objects sets **objdata* to reflect the first matching object that it finds.
  - On a CS_CLEAR operation, cs_objects clears all matching objects.

Retrieving "Current Connection" objects

- If an application has previously saved a CS_CURRENT_CONNECTION object, it can retrieve the current connection by:
  - Calling cs_objects with *objname–>object_type* as CS_CURRENT_CONNECTION, *lnlen* as CS_UNUSED, and *fnlen* as CS_UNUSED. cs_objects ignores the *last_name* and *first_name* fields of *objname*, and sets *objdata–>buffer* to the name of the current connection and *objdata–>buflen* to the length of this name.
  - Calling cs_objects with *objname–>object_type* as CS_CONNECTNAMEand *objname–>last_name* and *objname–>lnlen* as the newly retrieved connection name and name length. cs_objects sets *objdata* to the retrieved connection.

Warning! An application cannot call cs_objects(CS_SET) from within a completion callback routine.

See also cs_ctx_alloc

cs_prop_ssl_localid

Description

Specifies the path to the local ID (certificates) file.

Syntax

```
typedef struct _cs_sslid
{
      CS_CHAR    *identity_file;
      CS_CHAR    *identity_password;
} CS_SSLIDENTITY
```

Parameters

identity_file

provides a path to the file containing a digital certificate and the associated private key.

CS_GET only returns the *indentity_file* used, and only if it is set with CS_CONNECTION.

identity_password

used to decrypt the private key.

cs_set_convert

Description

Installs or retrieves a user-defined conversion routine.

Syntax

```
CS_RETCODE cs_set_convert(context, action, srctype,
        desttype, func)

CS_CONTEXT     *context;
CS_INT         action;
CS_INT         srctype;
CS_INT         desttype;
CS_CONV_FUNC   *func;
```

Parameters

context

A pointer to a CS_CONTEXT structure. A CS_CONTEXT structure defines a Client-Library application context.

action

One of the following symbolic values:

Value of action	cs_set_convert
CS_SET	Installs a conversion routine.
CS_GET	Retrieves the current conversion routine of this type.
CS_CLEAR	Clears the current conversion routine by replacing it with CS-Library's default conversion routine of this type.

srctype
The datatype of the source data for the conversion.

desttype
The datatype of the destination data.

func
A pointer to a CS_CONV_FUNC variable, which is a pointer to a custom conversion function. “Defining a custom conversion routine” on page 83 describes the prototype for a custom conversion function.

If a conversion routine is being installed, **func* points to the conversion routine that you wish to install.

If a conversion routine is being retrieved, cs_set_convert sets **func* to point to the currently installed conversion routine.

If a conversion routine is being cleared, pass **func* as NULL.

Note *func* represents a pointer to a pointer to a function. There are special requirements for passing this parameter. See the example code fragment under “Installing a custom conversion routine” on page 85.

Return value

cs_set_convert returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

The most common reason for a cs_set_convert failure is an invalid parameter.

Usage

- An application can install custom conversion routines to convert data between:
  - Standard Open Client or Open Server datatypes
  - Standard and user-defined datatypes
  - User-defined datatypes
- Once a custom routine is installed for a particular conversion, the client/server libraries call the custom routine transparently whenever a conversion of the specified type is required.
- A Client-Library or Server-Library application creates a user-defined datatype by declaring it:

```
typedef CS_SMALLINT          EMPLOYEE_ID;
```

Because the Open Client routines ct_bind and cs_convert use integer symbolic constants to identify datatypes, it is often convenient for an application to declare a type constant for a user-defined type. User-defined types must be defined as greater than or equal to CS_USERTYPE:

```
#define EMPLOYEE_ID_TYPE     CS_USERTYPE + 1;
```

To enable conversion between a user-defined type and standard CS-Library datatypes, an application can call cs_set_convert to install user-defined conversion routines for the new type.

- To clear a custom conversion routine, an application can call cs_set_convert with *action* as CS_CLEAR and *func* as NULL. cs_set_convert replaces the custom routine with CS-Library's default conversion routine of the appropriate type, if any.
- An application can call cs_setnull to define null substitution values for a user-defined type.

Defining a custom conversion routine

- A custom conversion routine is defined as follows:

```
CS_RETCODE CS_PUBLIC
 convfunc(context, srcfmt, srcdata,
          destfmt, destdata, destlen)
 CS_CONTEXT          *context;
 CS_DATAFMT          *srcfmt;
 CS_VOID             *srcdata;
 CS_DATAFMT          *destfmt;
 CS_VOID             *destdata;
 CS_INT              *destlen;
```

where:

- *context* is a pointer to a CS_CONTEXT structure.
- *srcfmt* is a pointer to a CS_DATAFMT structure describing the source data. *srcfmt→maxlength* describes the actual length, in bytes, of the source data.
- *srcdata* is a pointer to the source data.
- *destfmt* is a pointer to a CS_DATAFMT structure describing the destination data. *destfmt→maxlength* describes the actual length, in bytes, of the destination data space.
- *destdata* is a pointer to the destination data space.

- *destlen* is a pointer to an integer. If the conversion is successful, the custom routine should set **destlen* to the number of bytes placed in **destdata*.

- cs_config is the only CS-Library, Client-Library, or Server-Library function that can be called from within a custom conversion routine.
- The following table lists the legal return values for a custom conversion routine. CS-Library will raise a CS-Library error if any value other than CS_SUCCEED is returned. Other values should be returned to indicate error conditions, as described in Table 2-16.
  - If the conversion routine returns a value listed in Table 2-16 other than CS_SUCCEED, then the application receives a Client-Library or C S underscore -Library message that corresponds to the indicated error condition.
  - If the conversion routine returns a value that is not listed in Table 2-16, then the application receives an “Unknown return code” error message from Client-Library or CS-Library:

Table 2-16: Return values for a custom conversion routine

Return value	Indicates
CS_SUCCEED	Successful conversion.
CS_TRUNCATED	The conversion resulted in truncation.
CS_MEM_ERROR	A memory allocation failure has occurred.
CS_EBADXLT	Some characters could not be translated.
CS_ENOXLT	The requested translation is not supported.
CS_EDOMAIN	The source value is outside the domain of legal values for the datatype.
CS_EDIVZERO	Division by 0 is not allowed.
CS_EOVERFLOW	The conversion resulted in overflow.
CS_EUNDERFLOW	The conversion resulted in underflow.
CS_EPRECISION	The conversion resulted in loss of precision.
CS_ESCALE	An illegal scale value was encountered.
CS_ESYNTAX	The conversion resulted in a value which is not syntactically correct for the destination type.
CS_ESTYLE	The conversion operation was stopped due to a style error.

Installing a custom conversion routine

The following code demonstrates calling cs_set_convert to install a custom conversion routine, MyConvert, which converts from CS_CHAR to the user defined type indicated by MY_USER_TYPE. The code assumes that MyConvert is a a custom conversion routine that has been defined correctly. (See “Defining a custom conversion routine” on page 83.) The program variable *myfunc* is used to pass the address of the conversion routine.

```
#define MY_USER_TYPE (CS_USER_TYPE + 2)

CS_CONV_FUNC p_conv_func;

p_conv_func = MyConvert;
 if (cs_set_convert(context, CS_SET, CS_CHAR_TYPE, MY_USER_TYPE,
     &p_conv_func) != CS_SUCCEED)
 {
     fprintf(stdout, "cs_set_convert(MY_USER_TYPE) failed!\n");
     (CS_VOID)ct_exit(context, CS_FORCE_EXIT);
     (CS_VOID)cs_ctx_drop(context);
     exit(1);
 }
```

See also cs_convert, cs_manage_convert, cs_setnull, ct_bind

cs_setnull

Description

Defines a null substitution value to be used when binding or converting NULL data.

Syntax

```
CS_RETCODE cs_setnull(context, datafmt, buffer,
        buflen)

CS_CONTEXT      *context;
CS_DATAFMT      *datafmt;
CS_VOID         *buffer;
CS_INT          buflen;
```

Parameters

context

A pointer to a CS_CONTEXT structure. cs_setnull defines a null substitution value for this context.

datafmt

A pointer to a CS_DATAFMT structure describing the datatype for which a null substitution value is being defined.

buffer

A pointer to the null substitution value. **buffer*'s datatype must match *datafmt–>type*.

buflen

The length, in bytes, of **buffer*.

Return value

cs_set_null returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Common reasons for a cs_setnull failure include:

- A memory allocation error
- An invalid parameter

Usage

- If ANSI-style binds are in effect, CS-Library does not use null substitution values. To activate ANSI-style binds, an application sets the Client-Library property CS_ANSI_BINDS to CS_TRUE.
- When ANSI-style binds are not in effect and source data for a conversion is NULL, CS-Library sets the destination data to the predefined null substitution value for that destination type. For example, converting a NULL value of any type to a CS_CHAR destination results in an empty string.

- In a Client-Library application, null substitution values are defined at the context level. When a Client-Library connection is allocated, it picks up null substitution values from its parent context.
- When converting a NULL source value to a CS_CHAR or CS_BINARY destination variable, CS-Library first puts 0 bytes into the destination and then uses the *format* field of the CS_DATAFMT structure that describes the destination to determine whether to pad or null-terminate.
- To reinstate CS-Library's original default null substitution value for a particular datatype, an application can call cs_setnull with *buffer* as NULL.
- CS-Library and Client-Library use the following default null substitution values:

Table 2-17: Default null substitution values

Destination type	Null substitution value
CS_BINARY_TYPE	Empty array
CS_VARBINARY_TYPE	Empty array
CS_BIT_TYPE	0
CS_CHAR_TYPE	Empty string
CS_VARCHAR_TYPE	Empty string
CS_DATE_TYPE	4 bytes of zeros
CS_TIME_TYPE	4 bytes of zeros
CS_BIGDATETIME_TYPE	8 bytes of zeros
CS_BIGTIME_TYPE	8 bytes of zeros
CS_DATETIME_TYPE	8 bytes of zeros
CS_DATETIME4_TYPE	4 bytes of zeros
CS_TINYINT_TYPE	0
CS_SMALLINT_TYPE	0
CS_INT_TYPE	0
CS_DECIMAL_TYPE	0.0 (with default scale and precision)
CS_NUMERIC_TYPE	0.0 (with default scale and precision)
CS_FLOAT_TYPE	0.0
CS_REAL_TYPE	0.0
CS_MONEY_TYPE	$0.0
CS_MONEY4_TYPE	$0.0
CS_BOUNDARY_TYPE	Empty string
CS_SENSITIVITY_TYPE	Empty string
CS_TEXT_TYPE	Empty string
CS_UNITEXT_TYPE	Empty string
CS_IMAGE_TYPE	Empty array
CS_XML_TYPE	Empty string

See also cs_set_convert, cs_will_convert

cs_strbuild

Description Constructs native language message strings.

Syntax
```
CS_RETCODE cs_strbuild(context, buffer, buflen,
        resultlen, text, textlen
```

```
        [, formats, formatlen]
        [, arguments]);

CS_CONTEXT    *context;
CS_CHAR       *buffer;
CS_INT        buflen;
CS_INT        *resultlen;
CS_CHAR       *text;
CS_INT        textlen;
CS_CHAR       *formats;      /* Optional */
CS_INT        formatlen;     /* Optional */
<optional arguments>
```

Parameters

context

A pointer to a CS_CONTEXT structure.

buffer

A pointer to the space in which cs_strbuild places the finished message. Note that the finished message is not null-terminated. An application must use **resultlen* to determine the length of the message placed in **buffer*.

buflen

The length, in bytes, of the **buffer* data space.

resultlen

A pointer to an integer variable. cs_strbuild sets **resultlen* to the length, in bytes, of the string placed in **buffer*.

text

A pointer to the unfinished text of the message. The **text* string contains message text and placeholders for variables. A placeholder has the form %*integer*!, for example, %1!, %2!, and so forth. The integer indicates which argument to substitute for a particular placeholder. Arguments are numbered from left to right.

textlen

The length, in bytes, of **text*. If **text* is null-terminated, pass *textlen* as CS_NULLTERM.

formats

A pointer to a string containing one sprintf-style format specifier for each place holder in the **text* string.

formatlen

The length, in bytes, of **formats*. If **formats* is null-terminated, pass *formatlen* as CS_NULLTERM.

arguments

The values which will be converted to character according to the **formats* string and substituted into the **text* string to produce the message that is placed in **buffer*.

There must be one argument for each place holder. The first value corresponds to the first format and the %1! placeholder, the second value corresponds to the second format and the %2! placeholder, and so forth.

If insufficient arguments are supplied, cs_strbuild generates unpredictable results.

If too many arguments are supplied, the excess arguments are ignored.

Return value

cs_str_build returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- cs_strbuild builds a printable native-language message string from a text containing place holders for values, a format string containing information on the types and appearances of the values, and a variable number of arguments that represent the values.
- Parameters in error messages can occur in different orders in different languages. cs_strbuild allows an application to construct error messages in a sprintf-like fashion to ensure easy translation of error messages from one language to another.

  For example, consider an error message that informs the user of a misused keyword in a stored procedure. The message requires three arguments: the misused keyword, the line in which the keyword occurs, and the name of the stored procedure. In the U.S. English localization file, the message text appears as:

  ```
  The keyword '%1!' is misused in line %2! of stored
  procedure '%3!'.
  ```

  In the Spanish localization file, the same message appears as:

  ```
  En linea %2! de stored procedure '%3!', la palabra
  '%1!' esta mal usado!
  ```

  The cs_strbuild call for either of the above messages is:

  ```
  cs_strbuild(context, &mybuffer, buflength,
      &resultlength, messagetext, CS_NULLTERM,
      "%s, %d, %s", CS_NULLTERM,
  ```

```
keyword, linenum, sp_name);
```

The only difference is the content of *messagetext*.

- cs_strbuild format specifiers can be separated by other characters, or they can be adjacent to each other. This allows existing message strings in U.S. English to be used as format parameters. The first format specifier describes the %1! placeholder, the second describes the %2! placeholder, and so forth.

See also cs_dt_crack, cs_dt_info, cs_locale

cs_strcmp

Description Compares two strings using a specified sort order.

Syntax

```
CS_RETCODE cs_strcmp(context, locale, type, str1,
        len1, str2, len2, result)

CS_CONTEXT   *context;
CS_LOCALE    *locale;
CS_INT       type;
CS_CHAR      *str1;
CS_INT       len1;
CS_CHAR      *str2;
CS_INTl      len2;
CS_INT       *result;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

locale

A pointer to a CS_LOCALE structure. A CS_LOCALE structure contains locale information, including the collating sequence that cs_strcmp uses to define a sort order.

An application can call cs_locale with *type* as CS_LC_COLLATE or CS_SYB_SORTORDER to change the collating sequence in a CS_LOCALE structure.

locale can be NULL. If *locale* is NULL, cs_strcmp uses whatever localization information is defined in the *context* CS_CONTEXT structure. Localization information is always defined at the context level, because a CS_CONTEXT picks up default localization information when it is allocated.

type
The type of comparison to perform.

If *type* is CS_COMPARE, cs_strcmp performs a lexicographic comparison.

If *type* is CS_SORT, the values are compared as they would appear in a sorted list. It is possible for strings that are lexicographically equal to belong in different places in a sorted list.

str1
A pointer to the first string for the comparison.

len1
The length, in bytes, of **str1*. If **str1* is null-terminated, pass *len1* as CS_NULLTERM.

str2
A pointer to the second string for the comparison.

len2
The length, in bytes, of **str2*. If **str2* is null-terminated, pass *len2* as CS_NULLTERM.

result
A pointer to the result of the comparison. The following table lists the possible values for **result*:

Value of *result	Indicates
<0	*str1* is lexicographically less than *str2*, or *str1* appears before *str2* in a sorted list.
0	*str1* is lexicographically equal to *str1*, or *str1* is identical to *str2*.
>0	*str1* is lexicographically greater than *str2*, or *str1* appears after *str2* in a sorted list.

Return value

cs_strcmp returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- cs_strcmp sets **result* to indicate the result of the comparison.
- Some languages contain strings that are lexicographically equal, according to a specific sort order, but contain different characters. Although the strings are lexicographically equal, there is a standard order used when placing them into a sorted list.

An application can use cs_strcmp to compare strings either lexicographically or how they appear in a sorted list. For example, given a sort order that specifies that uppercase characters appear before lowercase characters in a sorted list:

- The strings “ABC” and “abc” are lexicographically equal.

  A call to cs_strcmp that compares “ABC” (as *str1*) and “abc” as (*str2*) with *type* as CS_COMPARE returns with *result* set to 0.

- “ABC” appears before “abc” in a sorted list.

  A call to cs_strcmp that compares “ABC” (as *str1*) and “abc” as (*str2*) with *type* as CS_SORT returns with *result* set to a value less than 0.

- cs_strcmp determines which sort order to use by examining **locale*, (or **context*, if *locale* is NULL).
  - To change the sort order in a CS_LOCALE structure, an application calls cs_locale with *type* as CS_LC_COLLATE or CS_SYB_SORTORDER.
  - To change the sort order in a CS_CONTEXT structure, an application must first set up a CS_LOCALE structure with the desired sort order and then call cs_config to set the CS_LOC_PROP property for the context.

See also cs_cmp, cs_locale, cs_config

cs_time

Description Retrieves the current date and time.

Syntax

```
CS_RETCODE cs_time(context, locale, buffer, buflen,
        outlen, daterec)

CS_CONTEXT    *context;
CS_LOCALE     *locale;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT        *outlen;
CS_DATEREC    *daterec;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

locale

A pointer to a CS_LOCALE structure. A CS_LOCALE structure contains locale information, including formatting information that cs_time uses to create a current datetime string.

locale can be NULL. If *locale* is NULL, cs_time uses whatever localization information is defined in the CS_CONTEXT structure indicated by *context*. Localization information is always defined at the context level, because a CS_CONTEXT picks up default localization information when it is allocated.

buffer

A pointer to the space in which cs_time will place a character string representing the current date and time.

buffer is an optional parameter and can be passed as NULL. If *buffer* is NULL, *daterec* must be supplied.

buflen

The length, in bytes, of **buffer*.

If *buffer* is supplied and *buflen* indicates that **buffer* is not large enough to hold the current datetime string, cs_time sets **outlen* to the length of the datetime string and returns CS_FAIL.

If *buffer* is NULL, pass *buflen* as CS_UNUSED.

outlen

A pointer to an integer variable.

cs_time sets **outlen* to the length, in bytes, of the current datetime string.

If the string is larger than *buflen* bytes, an application can use the value of **outlen* to determine how many bytes are needed to hold the string.

If *buffer* is NULL, pass *outlen* as NULL.

If an application does not care about return length information, it can pass *outlen* as NULL.

daterec

A pointer to a CS_DATEREC structure in which cs_time will place the current date and time. Note that cs_time does not set the *datemsecond* and *datetzone* fields of the CS_DATEREC structure.

See cs_dt_crack in this chapter.

daterec is an optional parameter and can be passed as NULL. If *daterec* is NULL, *buffer* must be supplied.

Return value

cs_time returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Common reasons for a cs_time failure include:

- An invalid parameter.
- *buflen* indicates that the **buffer* data space is not large enough to hold the formatted datetime string.

Usage

- cs_time returns the current date and time either in character string format or in a CS_DATEREC structure, or both.
- cs_time formats the date and time according to locale information contained in **context*.

See also

cs_config, cs_dt_crack, cs_dt_info, cs_locale

cs_validate_cb

Description

A Client-Library callback routine, registered through ct_callback.

Syntax

```
typedef struct _cs_sslcertfield
{
        CS_VOID      *value;
        CS_INT        field_id;
        CS_INT        length;
} CS_SSLCERT_FIELD;

typedef struct _cs_sslcert
{
        CS_INT field_count;
        CS_INT extension_count;
        CS_UINT                 start_date;
        CS_UINT                 end_date;
        CS_SSLCERT_FIELD    *fieldptr;
        CS_SSLCERT_FIELD    *extensionptr;
} CS_SSLCERT;

typedef CS_INT (CS_PUBLIC * CS_CERT_CB) PROTOTYPE ((
        CS_VOID          *user_data,
        CS_SSLCERT     *certptr,
        CS_INT             cert_count,
        CS_INT             valid
        ));
```

Parameters

certptr
A pointer to an array of CS_SSLCERT which has cert_count elements. On return from the callback, all memory used is freed.

Note The array is not null terminated.

fieldptr
A pointer to field_count elements.

extensionptr
A pointer extension_count elements.

cs_will_convert

Description

Indicates whether a specific datatype conversion is available in the Client/Server libraries.

Syntax

```
CS_RETCODE cs_will_convert(context, srctype, desttype,
        result)

CS_CONTEXT    *context;
CS_INT        srctype;
CS_INT        desttype;
CS_BOOL       *result;
```

Parameters

context
A pointer to a CS_CONTEXT structure.

srctype
A symbolic constant representing the datatype of the source data (for example, CS_BYTE_TYPE, CS_CHAR_TYPE, and so forth).

desttype
A symbolic constant representing the datatype of the destination data.

result
A pointer to a boolean variable. cs_will_convert sets **result* to CS_TRUE if the datatype conversion is supported and CS_FALSE if the datatype conversion is not supported.

Return value

cs_will_convert returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.

Returns	Indicates
CS_FAIL	The routine failed.

Examples

```
/*
 ** ex_display_column()
 */


CS_RETCODE CS_PUBLIC
 ex_display_column(context, colfmt, data, datalength,
      indicator)
 CS_CONTEXT           *context;
 CS_DATAFMT           *colfmt;
 CS_VOID              *data;
 CS_INT               datalength;
 CS_SMALLINT          indicator;
 {
    char         *null = "NULL";
    char         *nc   = "NO CONVERT";
    char         *cf   = "CONVERT FAILED";
    CS_DATAFMT   srcfmt;
    CS_DATAFMT   destfmt;
    CS_INT       olen;
    CS_CHAR      wbuf[MAX_CHAR_BUF];
    CS_BOOL      res;
    CS_INT       i;
    CS_INT       disp_len;

 if (indicator == CS_NULLDATA)
 {
     olen = strlen(null);
     strcpy(wbuf, null);
 }
 else
 {
     cs_will_convert(context, colfmt->datatype,
          CS_CHAR_TYPE, &res);

     if (res != CS_TRUE)
 {
        olen = strlen(nc);
        strcpy(wbuf, nc);
       }
       else
```

```
        {
         srcfmt.datatype  = colfmt->datatype;
         srcfmt.format    = colfmt->format;
         srcfmt.locale    = colfmt->locale;
         srcfmt.maxlength = datalength;

         destfmt.maxlength = MAX_CHAR_BUF;
         destfmt.datatype  = CS_CHAR_TYPE;
         destfmt.format    = CS_FMT_NULLTERM;
         destfmt.locale    = NULL;

         if (cs_convert(context, &srcfmt, data,
            &destfmt, wbuf, &olen) != CS_SUCCEED)
         {
            olen = strlen(cf);
            strcpy(wbuf, cf);
         }
         else
         {
            /*
            ** output length include null
            ** termination
            */
            olen -= 1;
         }

      }
   }
   fprintf(stdout, "%s", wbuf);

   disp_len = ex_display_dlen(colfmt);
   for (i = 0; i < (disp_len - olen); i++)
   {
      fputc(' ', stdout);
   }

      return CS_SUCCEED;
   }
```

Usage

- cs_will_convert allows an application to determine whether cs_convert or ct_bind/ct_fetch are capable of performing a specific conversion. When cs_convert is called to perform a conversion that it does not support, it returns CS_FAIL and generates a CS-Library error.

- cs_convert can convert between standard and user-defined datatypes. To enable these types of conversions, an application must install custom conversion routines through cs_set_convert. If a custom routine is supplied for a conversion, cs_will_convert indicates that the conversion is supported.

Datatype conversion chart

A chart listing the datatype conversions that cs_convert supports is included on the manual page for cs_convert. (See Table 2-3 on page 30.)

See also

cs_convert, cs_set_convert, cs_setnull

CHAPTER 3

Bulk-Library

This chapter introduces Bulk-Library:

Overview of Bulk-Library

Bulk-Library/C provides routines that allow Client-Library and Server-Library applications to use the Adaptive Server Enterprise bulk-copy interface.

The Adaptive Server Enterprise bulk copy interface allows high-speed transfer of data between a client application's program variables and the server's database tables. It provides an alternative to the use of the SQL insert and select commands to transfer data.

Administrators can perform bulk copy using the bcp utility; programmers can use Bulk-Library to create customized bulk-copy tools. Bulk-Library also provides the necessary routines to enable bulk-copy support in an Open Server gateway application.

Bulk copy of encrypted columns is supported if Adaptive Server Enterprise supports encrypted columns.

Note The Bulk-Library/C routines are for use with Open Client Client-Library and Open Server Server-Library applications. DB-Library™ provides its own bulk-copy interface, which is documented in the *Open Client DB-Library/C Reference Manual*.

Client-side and server-side routines

Bulk-Library contains client-side and server-side routines.

Client-side Bulk-Library routines

Client-side routines allow Client-Library programmers to execute bulk-copy commands from their programs. Client-side routines allow a program to:

- Transmit bulk-copy data to the remote server for database table population
- Extract the contents of a database table into program memory

Server-side Bulk-Library routines

Server-side routines are used with Open Server. Open Server programmers can use these routines together with the client-side routines to allow bulk-copy transfers through an Open Server gateway. A gateway server uses the client-side routines to obtain bulk-copy data from the remote server and server-side routines to forward the data to its own client. Any routine that requires a SRV_PROC (Open Server thread-control structure) pointer as an argument is a server-side routine.

The server-side Bulk-Library routines require the application to be linked with Server-Library and must be used together with the client-side routines.

Header files

The header file *bkpublic.h* contains Bulk-Library definitions and is required in all application source files that contain calls to Bulk-Library routines.

Client-Library applications that call Bulk-Library routines need to include only *bkpublic.h*, since *bkpublic.h* includes *ctpublic.h*. No harm is done if the application includes both files.

Gateway Open Server applications that call Bulk-Library routines need to include *bkpublic.h* in addition to the other include files required by Server-Library. *bkpublic.h* does not include any Open Server header files.

Linking with Bulk-Library

On most platforms, Bulk-Library is a separate library file and must be specified on the link line for the application. See the *Open Client and Open Server Programmers Supplement* for compiling and linking instructions on your platform.

The CS_BLKDESC structure

All bulk-copy operations performed with Bulk-Library calls require a CS_BLKDESC structure. This structure is also called the *bulk-descriptor structure*. The bulk-descriptor structure is a hidden structure that controls a particular bulk-copy operation.

Applications allocate a bulk-descriptor structure with blk_alloc on page 118 and free the bulk descriptor's memory with blk_drop on page 141. The structure's internals are not documented, but the properties of the structure can be retrieved and modified with the blk_props on page 149 routine.

All Bulk-Library routines except for blk_alloc require a valid bulk-descriptor structure pointer as an input parameter.

The bulk-descriptor structure is considered a child structure of Client-Library's connection structure. Bulk-copy operations require the connection to interact with the remote server.

Bulk-Library client programming

Client-side Bulk-Library routines provide bulk-copy functionality to Client-Library programs. A Client-Library programmer may find bulk-copy useful if the application under development must exchange data with a non-database application, load data into a new database, or move data from one database to another.

A Client-Library application can call Bulk-Library routines to copy data either into a database table or out from a database table.

- Bulk-copy-in operations move data from the client machine into a database table and are typically used for database table population. For bulk copies into the database, Bulk-Library transmits tabular data over the network in its "raw" form. Bulk copies into the database can be considerably faster than embedding the data in equivalent SQL insert statements.
- Bulk-copy-out operations move data from a database table to the client program's memory space and are typically used for data extracts. For data extracts, bulk copy offers no performance advantage over the equivalent SQL select statements. However, the Bulk-Library interface may be more convenient for programmers.

Note Errors resulting from client-side Bulk-Library routines are reported as Client-Library errors. Applications should install a Client-Library message callback to handle these errors or handle them inline with ct_diag.

Bulk-copy-in operations

An application can call Bulk-Library routines to copy data from program variables into a database table.

When copying into a database, the chief advantage of bulk copy over the SQL insert alternative is speed.

When copying data into a non-indexed table, the *high speed* version of bulk copy is used. Adaptive Server Enterprise performs no data logging during high-speed transfers. If the system fails before the transfer is complete, no new data will remain in the database. Because high-speed transfer affects the recoverability of the database, it is enabled only when the Adaptive Server Enterprise option select into/bulkcopy has been turned on. An application can call the Adaptive Server Enterprise system procedure sp_dboption to turn this option on or use the Client-Library connection property CS_BULK_LOGIN.

If the select into/bulkcopy option is not turned on and a user tries to copy data into a table that has no indexes, Adaptive Server Enterprise generates an error message.

After a bulk-copy operation is complete, the System Administrator should dump the database to ensure its future recoverability.

When copying data into an indexed table, a slower version of bulk copy is automatically used, and row inserts are logged.

The bulk-copy-in process

A typical application follows these steps to perform a bulk-copy-in operation:

1 Initializes the application in the same way as for a Client-Library application and sets up Client-Library error handling. Bulk-Library reports errors generated by calls to client-side routines as Client-Library messages.

2 Allocates the connection structure to be used.

3 Calls ct_con_props to set the necessary properties to connect to the target server. In addition, the application must set the CS_BULK_LOGIN property to CS_TRUE to enable the connection to perform bulk copies.

Note Programmers can often tune the Tabular Data Stream™ (TDS) packet size to increase throughput. A packet size larger than the default usually increases performance. First, make sure that the Adaptive Server Enterprise is configured to accept a larger TDS packet size, then set the CS_PACKET_SIZE connection property in your application. See the *Adaptive Server Enterprise System Administration Guide* for details on increasing the allowable network packet size and the *Open Client Client-Library/C Reference Manual* for details on connection properties.

---

4 Calls ct_connect to open the connection.

5 Calls blk_alloc to allocate a bulk-descriptor structure.

6 Calls blk_init to initialize the bulk-copy operation.

7 For each column in the target table, the application:

- (Optional) Calls blk_describe, which returns a target column's description, allowing the application determine the column's datatype or size.
- (Optional) Calls blk_default, which returns a column's default value, if a default is defined by the table schema. An application can call blk_bind with **datalen* as 0 to indicate that the bulk-copy-in operation should use a column's default value.
- Calls blk_bind to bind the variable to the target column. If data for the column will be transferred using blk_textxfer, the application must call blk_bind with *buffer* as NULL.

Columns can be bound either to scalar variables or to arrays. When columns are bound to scalar variables, each call to blk_rowxfer_mult transfers column values for a single row from the bound variables into the database. For array binding, an array is bound to each column, and multiple rows are transferred by each call to blk_rowxfer_mult. In either case, the application also binds *indicator* and *datalen* variables to the column as well. These are used to indicate the condition of the data to be transferred.

The discussion in this chapter assumes that array binding is not in effect. See blk_bind in Chapter 4, "Bulk-Library Routines."

8 Transfers the data.

While data remains to be transferred, the application places data into the program variables that are bound to the table columns, then calls blk_rowxfer_mult to transfer the row.

Before each call to blk_rowxfer_mult, for each bound column, the application sets *datalen* and *indicator* values to specify what value should be inserted:

datalen value	indicator value	Result
> 0	Any (is ignored).	blk_rowxfer_mult reads *datalen* bytes from *buffer* as the column value.
0	0	The column's default value, if available, is inserted. If no default is available, NULL is inserted.
0	-1	NULL is inserted.

If the row contains columns whose data is being transferred in chunks, the application calls blk_textxfer in a loop for each column. Data being transferred via blk_textxfer must reside at the end of the row, following any bound columns.

The application can call blk_done(CS_BLK_BATCH), if needed, to send a batch of rows. This call instructs the Adaptive Server Enterprise to permanently save all rows transferred since the application's last blk_done call.

9 Calls blk_done(CS_BLK_ALL) to send the last batch of rows and indicate that the bulk-copy operation is complete.

10 Calls blk_drop to deallocate the bulk-descriptor structure.

Note An application can call blk_bind between calls to blk_rowxfer_mult to specify a different program variable address or length.

Program structure for bulk-copy-in operations

Most applications use a program structure similar to the following pseudocode to perform a bulk-copy-in operation:

```
ct_con_props to set connection properties
 ct_connect to open the connection
 blk_alloc to allocate a CS_BLKDESC
 blk_init to initiate the bulk copy

for each column
       (optional: blk_describe to get a description of
            the column)
       (optional: blk_default to get the column's default
            value)
        blk_bind to bind the column to a program
            variable, or to mark the column for transfer
            via blk_textxfer
 endfor

while there's data to transfer
        if it's time to save a batch of rows
            blk_done(CS_BLK_BATCH)
        endif
        copy row values to program variables
        call blk_rowxfer_mult to transfer the row data

      if data is being transferred via blk_textxfer
            for each column to transfer
                 while there's data for this column
                     blk_textxfer to tranfer a chunk of data
                 endwhile
            endfor
        endif
 endwhile
blk_done(CS_BLK_ALL)
 blk_drop to deallocate the CS_BLKDESC
```

Bulk-copy-out operations

The bulk-copy-out process reads rows from the server and places the column values into program variables.

The bulk-copy-out process

A typical application follows these steps to perform a bulk-copy-out operation:

1 Calls ct_con_props to set the required properties to open the connection.

2 Calls ct_connect to open the connection.

3 Calls blk_alloc to allocate a bulk-descriptor structure.

4 For each column of interest, the application:

- (Optional) Calls blk_describe to retrieve a column's description. This step is necessary if an application lacks information about a column's datatype or size.
- (Optional) Calls blk_bind to bind a program variable to the source column. If the data for a column will be transferred via blk_textxfer, call blk_bind with **buffer* as NULL.

  Columns can be bound either to scalar variables or to arrays. When columns are bound to scalar variables, each call to blk_rowxfer_mult transfers column values for a single row into the bound variables into the database. For array binding, an array is bound to each column, and multiple column values are transferred into each array by each call to blk_rowxfer_mult.

  The discussion in this chapter assumes that array binding is not used. See blk_bind in Chapter 4, "Bulk-Library Routines"

5 Transfers the data by calling blk_rowxfer_mult in a loop:

  The application calls blk_rowxfer_mult repeatedly to transfer each row to program variables until blk_rowxfer_mult returns CS_END_DATA.

  If the row contains columns whose data is transferred in chunks, the application calls blk_textxfer in a loop for each column. Data being transferred via blk_textxfer must reside at the end of the row, following any bound columns.

For example, suppose an application bulk-copies columns 1, 3, 5, 7, and 9 and must call blk_textxfer to copy columns 7 and 9. The application calls blk_bind once for each column, passing *buffer* as NULL for columns 7 and 9. After calling blk_rowxfer_mult to transfer a row from the table, the application must call blk_textxfer in a loop to copy the data for column 7 and then call blk_textxfer in another loop to copy the data for column 9.

6 Calls blk_done(CS_BLK_ALL) to indicate that the bulk-copy operation is complete.

7 Calls blk_drop to deallocate the bulk-descriptor structure.

Note An application can call blk_bind between calls to blk_rowxfer_mult to specify different program variable address or length.

Program structure for bulk-copy-out operations

Most applications use a program structure similar to the following pseudocode to perform a bulk-copy-out operation:

```
ct_con_props to set connection properties
 ct_connect to open the connection
 blk_alloc to allocate a CS_BLKDESC
 blk_init to initiate the bulk copy
 for each column of interest
          (optional: blk_describe to get a description of
                the column)
          blk_bind to either bind the column to a program
                variable or to indicate that blk_textxfer will
                be used to transfer data for the column.
 endfor
while there's data to transfer
          call blk_rowxfer_mult to transfer the row data
          pull data from program variables to a permanent
                location, if desired.
          if data is being transferred via blk_textxfer
                for each column to transfer
                   while there's data for this column
                      blk_textxfer to tranfer a chunk of data
                   endwhile
                endfor
          endif
 endwhile
 blk_done(CS_BLK_ALL)
 blk_drop to deallocate the CS_BLKDESC
```

Copying to and from Secure Adaptive Server Enterprise

Each row in a Secure Adaptive Server Enterprise table has a sensitivity column, which contains the sensitivity label for the row. Secure Adaptive Server Enterprise uses sensitivity labels to mediate access to data.

When bulk copying into or from a Secure Adaptive Server Enterprise table, an application can choose whether or not to include the table's sensitivity column in the bulk-copy operation.

To include the sensitivity column, an application sets the BLK_SENSITIVITY_LBL property to CS_TRUE. BLK_SENSITIVITY_LBL has a default value of CS_FALSE, which means that by default the sensitivity column is not included.

Users copying into the sensitivity column must have the bcpin_labels_role activated on Secure Adaptive Server Enterprise. If a user does not have this role, the bulk-copy operation fails. See your Secure Adaptive Server Enterprise documentation for more information on setting this role.

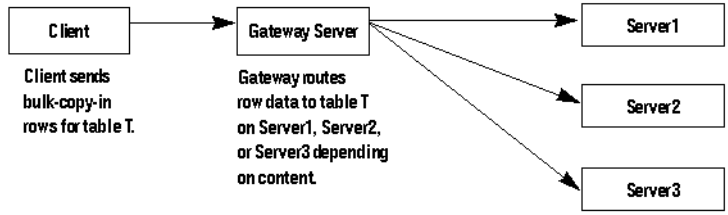
Bulk-Library gateway programming

The server-side Bulk-Library routines are designed to be used in gateways in conjunction with the client-side routines. Note that Open Server applications must have available a valid CS_CONNECTION structure (set up with Client-Library calls) to call Bulk-Library routines.

Open Server provides bulk-copy functionality that allows gateway Open Server applications to filter bulk-copy data. A gateway Open Server can examine each row of a bulk-copy operation and implement any of the following filters:

- Discard certain rows while keeping others,
- Send all rows to the remote server, or
- Route bulk-copy requests to multiple remote servers based on the row content, as shown in the diagram below.

Figure 3-1: Gateway routing bulk-copy requests



A gateway's client can issue two types of bulk requests, a *TDS text/image insert request* or a *TDS bulk-copy request*. In the case of a TDS text/image insert, the client simply wishes to send a text or image stream. In the case of a TDS bulk-copy request, the client is actually initiating a bulk-copy request. In both cases, the request handling involves processing both language (SRV_LANGUAGE) events and bulk (SRV_BULK) events.

An Open Server application processes both requests using two event handlers: SRV_LANGUAGE and SRV_BULK. Inside the SRV_LANGUAGE event handler, the application determines which kind of bulk request has been issued by the client and records this information internally. In addition, if the request is for bulk copy, the application allocates and initializes a bulk-descriptor structure. Inside the SRV_BULK handler, the application retrieves the request type and then processes the data accordingly.

The discussion in this section assumes that the gateway application is intended to accept both bulk-copy insert requests and text/image insert requests. For a description of how to handle text/image insert commands only, see the "Text and Image" topics page in the *Open Server Server-Library/C Reference Manual*.

Note Bulk-Library reports errors resulting from calls to server-side routines as Server-Library errors. Applications that call server-side Bulk-Library routines should install a Server-Library error handler to receive notification of these errors.

Inside the SRV_LANGUAGE event handler

If you intend for your gateway application to handle either type of bulk request, you must code the SRV_LANGUAGE event handler to parse for the phrase "insert bulk" or "writetext bulk." These phrases indicate the following:

- The phrase "insert bulk" indicates the initiation of a bulk-copy request; the request handling will be started in the language handler and finished in the SRV_BULK handler.
- The phrase "writetext bulk" indicates that the client will issue a stream of text or image bytes to be handled in the SRV_BULK event handler.

"Insert Bulk" requests

The text of an "insert bulk" language request looks like this:

```
insert bulk tablename [with nodescribe]
```

where "with nodescribe" is optional.

In response, the SRV_LANGUAGE event handler should:

1 Record the bulk type internally by calling srv_thread_props with *cmd* set to CS_SET, *property* set to SRV_T_BULKTYPE, and *bufp* pointing to a value of SRV_BULKLOAD.

2 Continue parsing to extract the table name, which is an argument to the blk_init routine. The table name is in the form of "*database.owner.tablename*", without slice information. If a slice is used for the bulk insert command, the colon and slice number must be removed from the table name.

3 Allocate a bulk-descriptor structure, CS_BLKDESC, with a call to blk_alloc.

4 Initialize the client half of the exchange with a call to blk_init.

5 If "with nodescribe" is specified, it means that this data is part of a batch, and the table into which the bulk data will be loaded has already been described. The application need not call blk_srvinit a second time.

   If "with nodescribe" is not specified, initialize the server half of the exchange with a call to blk_srvinit.

"Writetext Bulk" requests

The text of a "writetext bulk" language request looks like this:

```
writetext bulk dbname.tblname.colname textptr
[timestamp=timestamp] [with log]
```

where the timestamp and logging indicator are optional.

In response, the SRV_LANGUAGE event handler should:

1 Record the bulk type internally by calling srv_thread_props with *cmd* set to CS_SET, *property* set to SRV_T_BULKTYPE, and *bufp* pointing to a value of SRV_TEXTLOAD, SRV_IMAGELOAD, or SRV_UNITEXTLOAD.

2 Continue parsing to extract the object name, which is generally of the form "*dbname.tblname.colname*". This name can then be stored in the name and namelen fields of a CS_IODESC structure, which can later be used in the SRV_BULK event handler as an argument to ct_data_info, if the data stream is being passed on to a server in a gateway application.

3 Continue parsing to extract the text pointer, which will appear as a large hexadecimal number. Once converted from a character string to an actual CS_BINARY value, the text pointer and its length are stored in the textptr and textptrlen fields of the CS_IODESC structure.

4 Continue parsing to extract the timestamp, which, if present, will appear as "timestamp = *large_hexadecimal_number*". Once converted from a character string to an actual CS_BINARY value, the timestamp and its length can be stored in the timestamp and timestamplen fields of the CS_IODESC structure.

5 Finally, parse to extract the logging indicator, which, if present, will appear as "with log". If this indicator is present, the log_on_update field of the CS_IODESC structure should be set to CS_TRUE.

Inside the SRV_BULK event handler

Inside the SRV_BULK event handler, the application must respond to the bulk request that triggered the handler. However, its response depends on which type of bulk request the client issued. The application retrieves the request type by calling srv_thread_props with cmd set to CS_GET and *property* set to SRV_T_BULKTYPE.

If the request type is SRV_TEXTLOAD, SRV_IMAGELOAD, or SRV_UNITEXTLOAD, the application reads the text or image data from the client in chunks, using the srv_text_info and srv_get_text routines. For details, see the "Text and Image" topics page in the *Open Server Server-Library/C Reference Manual*.

If the request type is SRV_BULKLOAD, the application processes the bulk-copy rows using a combination of client-side and server-side routines. To process the bulk-copy rows, the SRV_BULK event handler should:

1 Call blk_rowalloc to allocate a CS_BLK_ROW structure.

   The CS_BLK_ROW structure is a hidden structure that holds formatted bulk-copy rows sent from the client.

2 Call blk_getrow to retrieve the formatted row from the client. This call retrieves all column data except columns of type text, image, sensitivity, or boundary. The gateway can process these later. If the row contains text, image, sensitivity, or boundary data, blk_getrow returns CS_BLK_HASTEXT. Otherwise, it returns CS_SUCCEED. If there are no more rows, the bulk-copy operation is complete and blk_getrow returns CS_END_DATA.

3 If the gateway must examine the row content (for example, to route rows to particular remote servers or reject data), it calls blk_colval to examine the value of each column in the bulk row.

4 Call the client-side routine blk_sendrow to send the formatted rows to the remote server.

5 If an incoming bulk row contains text, image, sensitivity, or boundary data, the server portion of the gateway calls blk_gettext to retrieve the row's text, image, sensitivity, or boundary portion. The handler calls the client-side routine blk_sendtext to send it on to the remote server.

6 Call blk_rowdrop to deallocate the CS_BLK_ROW structure allocated by blk_rowalloc.

7 Call the client-side routine blk_done to indicate that the batch or bulk-copy operation is complete.

8 Call blk_drop to deallocate the bulk-descriptor structure.

Example

The Open Server sample program *ctos.c* includes code to process bulk-copy requests.

CHAPTER 4

Bulk-Library Routines

This chapter contains a reference page for each Bulk-Library routine.

blk_alloc

Description

Allocates a CS_BLKDESC structure.

Syntax

CS_RETCODE blk_alloc(connection, version, blk_pointer)

```
CS_CONNECTION   *connection;
CS_INT          version;
CS_BLKDESC      **blk_pointer;
```

Parameters

connection

A pointer to a CS_CONNECTION structure that has been allocated with ct_con_alloc and opened with ct_connect. A CS_CONNECTION structure contains information about a particular client/server connection.

The connection must not have any pending results.

version

The intended version of Bulk-Library behavior. During initialization, *version*'s value is checked for compatibility with Client-Library's version level. *version* can take the following values:

Value	Meaning	Compatible Client-Library version level(s)
BLK_VERSION_100	Version 10.0 behavior	CS_VERSION_110, CS_VERSION_100
BLK_VERSION_110	Version 11.0 behavior	Same as BLK_VERSION_100
BLK_VERSION_120	Version 12.0 behavior	Same as BLK_VERSION_100, 110
BLK_VERSION_125	Version 12.5 behavior	Same as BLK_VERSION_100, 110, 120
BLK_VERSION_150	Version 15.0 behavior	Same as BLK_VERSION_100, 110, 120, 125
BLK_VERSION_155	Version 15.5 behavior	Same as BLK_VERSION_100, 110, 120, 125, 150

Note BLK_VERSION_100 can only be used with Open Client and Open Server versions 11.x and higher, regardless of whether the context/ctlib is initialized to CS_VERSION_100 or CS_VERSION_110.

The application's Client-Library version level is determined by the call to ct_init that initializes the connection's parent context structure.

blk_pointer
The address of a pointer variable. blk_alloc sets **blk_pointer* to the address of a newly allocated CS_BLKDESC structure.

In case of error, blk_alloc sets **blk_pointer* to NULL.

Return value

blk_alloc returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

The most common reason for a blk_alloc failure is a lack of adequate memory.

Examples

```
/*
 ** BulkCopyIn()
 ** Ex_tabname is globally defined.
 */
 CS_STATIC CS_RETCODE
 BulkCopyIn(connection)
 CS_CONNECTION   *connection;
 {
      CS_BLKDESC     *blkdesc;
      CS_DATAFMT     datafmt;     /* variable descriptions */
      Blk_Data       *dptr;       /* data for transfer */
      CS_INT         datalen[5];  /* variable data length */
      CS_INT         len;
      CS_INT         numrows;

      /*
      ** Ready to start the bulk copy in now that all the
      ** connections have been made and have a table name.
      ** Start by getting the bulk descriptor and
      ** initializing.
      */
      if (blk_alloc(connection, BLK_VERSION_100, &blkdesc)
            != CS_SUCCEED)
      {
            ex_error("BulkCopyIn: blk_alloc() failed");
            return CS_FAIL;
 }
  if (blk_init(blkdesc, CS_BLK_IN,
            Ex_tabname, strlen(Ex_tabname)) == CS_FAIL)
 {
            ex_error("BulkCopyIn: blk_init() failed");
```

```
            return CS_FAIL;
    }
    /*
    ** Bind the variables to the columns and send the rows,
    ** and then clean up.
    */
    ...CODE DELETED.....

    return CS_SUCCEED;
    }
```

Usage

- A CS_BLKDESC structure, also called a *bulk-descriptor structure*, is the control structure for sending and receiving bulk-copy data. It is a hidden structure that contains information about a particular bulk-copy operation.
- Before calling blk_alloc, an application must call the Client-Library routines ct_con_alloc and ct_connect to allocate a CS_CONNECTION structure and open the connection.
- blk_alloc must be the first routine called in a bulk-copy operation.
- Multiple CS_BLKDESC and CS_COMMAND structures can be allocated on a connection, but only one CS_BLKDESC or CS_COMMAND structure can be active at a time. See blk_init on page 146 in this chapter.
- To deallocate a CS_BLKDESC structure, an application can call blk_drop.

See also

blk_drop, blk_init, ct_con_alloc, ct_connect

blk_bind

Description

Bind a program variable to a database column.

Syntax

```
CS_RETCODE blk_bind(blkdesc, colnum, datafmt, buffer,
        datalen, indicator)

CS_BLKDESC    *blkdesc;
CS_INT        colnum;
CS_DATAFMT    *datafmt;
CS_VOID       *buffer;
CS_INT        *datalen;
CS_SMALLINT   *indicator;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

colnum

The number of the column to bind to the program variable. The first column in a table is column number 1, the second is number 2, and so forth. Only visible columns are counted.

Note Setting the ct_options parameter to CS_OPT_HIDE_VCC or CS_OPT_SHOW_FI influences which columns are visible.

If CS_OPT_HIDE_VCC is set to CS_TRUE, Virtual Computed Columns (VCC) are not visible and not represented by column numbers in blk_bind. Similarly, if CS_OPT_SHOW_FI remains as CS_FALSE, Functional Indexes (FI) are not visible and also not represented by column numbers in blk_bind.

See the *Open Client Client-Library/C Reference Manual*.

---

datafmt

A pointer to the CS_DATAFMT structure that describes the program variable to bind to the column.

Table 4-1 lists the fields in **datafmt* that are used by blk_bind and contains general information about the fields. blk_bind ignores fields that it does not use:

Table 4-1: Fields in the CS_DATAFMT structure for blk_bind

Field name	When used	Sets the field to
name	Not used.	Not applicable.
namelen	Not used.	Not applicable.
datatype	Always.	A type constant (CS_xxx_TYPE) representing the datatype of the program variable. All type constants listed on the "Types" topics page in the *Open Client Client-Library/C Reference Manual* are valid. Open Client user-defined types are not valid. blk_bind supports a wide range of type conversions, so *datatype* can be different from the column's type. For instance, by specifying a variable type of CS_FLOAT_TYPE, a *money* column can be bound to a CS_FLOAT program variable. blk_rowxfer_mult on page 160 or blk_rowxfer on page 156 perform appropriate conversions when transferring data. For a list of the data conversions provided by Client-Library, see cs_convert on page 27 in Chapter 2, "CS-Library Routines." If *datatype* is CS_BOUNDARY_TYPE or CS_SENSITIVITY_TYPE, the **buffer* program variable must be of type CS_CHAR.

format	When binding to character or binary-type destination variables during copy-out operations; otherwise, CS_FMT_UNUSED.	A bit-mask of the following destination types and related symbols:
		• For character and text destination types:
		  • CS_FMT_NULLTERM to null-terminate data.
		  • CS_FMT_PADBLANK to pad to full variable length with spaces.
		• For character, binary, text, and image destination types:
		  • CS_FMT_PADNULL to pad to full variable length with nulls.
		• For any destination type:
		  • CS_FMT_UNUSED if no format information is provided.
		• When using array binding, the only format flag for bulk-copy-in operations is CS_BLK_ARRAY_MAXLEN. See "Array binding" on page 131.

Field name	When used	Sets the field to
maxlength	When binding to a variable length datatype. When binding to a fixed-length datatype, *maxlength* is ignored.	The maximum length of the **buffer* program variable. When binding character or binary variables, *maxlength* must describe the total maximum length of the program variable, including any space required for special terminating bytes, such as a null terminator. During a bulk-copy-in operation, *maxlength* specifies the maximum length of the data that will be copied from the **buffer* program variable. During a bulk-copy-out operation, *maxlength* is the length of the **buffer* program variable.
scale	Only when binding to numeric or decimal variables.	The scale of the program variable. If the source data is the same type as the destination, then *scale* can be set to CS_SRC_VALUE to indicate that the destination should pick up its value for *scale* from the source data. *scale* must be less than or equal to *precision*.
precision	Only when binding numeric or decimal destinations.	The precision of the program variable. If the source data is the same type as the destination, then *precision* can be set to CS_SRC_VALUE to indicate that the destination should pick up its value for *precision* from the source data. *precision* must be greater than or equal to *scale*.
status	Not used.	Not applicable.

Field name	When used	Sets the field to
count	Always.	*count* is the number of rows to transfer per blk_rowxfer_mult on page 160 or blk_rowxfer on page 156 call. If *count* is greater than 1, array binding is considered to be in effect. During a bulk-copy-out operation, if *count* is larger than the number of available rows, only the available rows are copied. *count* must have the same value for all columns being transferred, with one exception: An application can intermix counts of 0 and 1. This is because when *count* is 0, 1 row is transferred.
usertype	Not used.	Not applicable.
locale	If supplied, *locale* is used. Otherwise, default localization applies.	A pointer to a CS_LOCALE structure containing locale information for the **buffer* program variable.

buffer

The address of the program variable to be bound to the column specified by *colnum*.

For a bulk-copy-in operations, **buffer* is the program variable from which blk_rowxfer_mult copies the data.

For bulk-copy-out operations, *buffer** is the program variable in which blk_rowxfer_mult places the copied data. If *datafmt–>maxlength* indicates that **buffer* is not large enough to hold the copied data, blk_rowxfer_mult truncates the data at row transfer time. If this occurs, Bulk-Library sets **indicator* to the actual length of the available data.

A NULL *buffer* indicates that data for the column will be transferred using the blk_textxfer routine.

datalen

A pointer to the length, in bytes, of the **buffer* data.

For bulk-copy-in operations:

- If **buffer* is not NULL, **datalen* represents the actual length of the data contained in the **buffer* program variable. An application must set this length before calling blk_rowxfer_mult or blk_rowxfer to transfer rows. In case of variable-length data, the length may be different for each row. If the data is fixed-length, **datalen* can be CS_UNUSED, except for array binding. If **datalen* is 0, the value of **indicator* is used to determine whether the column's default value or a NULL should be inserted. See Table 4-2 on page 129 for details.
- If **buffer* is NULL (indicating that the data will be transferred with blk_textxfer), **datalen* indicates the total length of the value to be transferred.

For bulk-copy-out operations:

- **datalen* represents the actual length of the data copied to **buffer*. blk_rowxfer_mult or blk_rowxfer sets **datalen* each time it is called to transfer a row.
- Since blk_rowxfer_mult or blk_rowxfer sets *datalen* each time it is called to transfer a row, the *datalen* parameter must remain local to the function calling blk_bind() and blk_rowxfer(), or blk_rowxfer_mult(). Failure to do so causes invalid results.

indicator

A pointer to a CS_INT variable, or for array binding, an array of CS_INT. At row-transfer time, blk_rowxfer_mult or blk_rowxfer read the indicator's contents to determine certain conditions about the bulk-copy data.

Return value

blk_bind returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

blk_bind returns CS_FAIL if the application has not called blk_init to initialize the bulk-copy operation.

Examples

```
/*
 ** BulkCopyIn()
 ** BLKDATA and DATA_END are defined in the bulk copy
```

```
** example program.
*/

CS_STATIC CS_RETCODE

BulkCopyIn(connection)
CS_CONNECTION   *connection;
{
     CS_BLKDESC    *blkdesc;
     CS_DATAFMT    datafmt;    /* variable descriptions */
     Blk_Data      *dptr;      /* data for transfer */
     CS_INT        datalen[5]; /* variable data length */
     CS_INT        len;
     CS_INT        numrows;

   /*
   ** Ready to start the bulk copy in now that all the
   ** connections have been made and have a table name.
   ** Start by getting the bulk descriptor initializing.
   */
   ...CODE DELETED.....

   /*
   ** Bind the variables to the columns and
   ** transfer the data.
   */
   datafmt.locale = 0;
   datafmt.count = 1;
   dptr = BLKDATA;
   while (dptr->pub_id != DATA_END)
   {
        datafmt.datatype = CS_INT_TYPE;
        datafmt.maxlength = sizeof(CS_INT);
        datalen[0] = CS_UNUSED;

       if (blk_bind(blkdesc, 1, &datafmt, &dptr->pub_id,

            &datalen[0], NULL) != CS_SUCCEED)
        {
            ex_error("BulkCopyIn: blk_bind(1) failed");
            return CS_FAIL;
        }
        datafmt.datatype = CS_CHAR_TYPE;
        datafmt.maxlength = MAX_PUBNAME - 1;
        datalen[1] = strlen(dptr->pub_name);
        if (blk_bind(blkdesc, 2, &datafmt, dptr->pub_name,
             &datalen[1], NULL) != CS_SUCCEED)
```

```
            {
                ex_error("BulkCopyIn: blk_bind(2) failed");
                return CS_FAIL;
            }
            datafmt.maxlength = MAX_PUBCITY - 1;
            datalen[2] = strlen(dptr->pub_city);
            if (blk_bind(blkdesc, 3, &datafmt, dptr->pub_city,
              &datalen[2], NULL) != CS_SUCCEED)
            {
                ex_error("BulkCopyIn: blk_bind(3) failed");
                return CS_FAIL;
            }
            datafmt.maxlength = MAX_PUBST - 1;
            datalen[3] = strlen(dptr->pub_st);
            if (blk_bind(blkdesc, 4, &datafmt, dptr->pub_st,
                 &datalen[3], NULL) != CS_SUCCEED)
            {
                ex_error("BulkCopyIn: blk_bind(4) failed");
                return CS_FAIL;
            }
            datafmt.maxlength = MAX_BIO - 1;
            datalen[4] = strlen((char *)dptr->pub_bio);
            if (blk_bind(blkdesc, 5, &datafmt, dptr->pub_bio,
                 &datalen[4], NULL) != CS_SUCCEED)
           {
               ex_error("BulkCopyIn: blk_bind(5) failed");
               return CS_FAIL;
           }
           if (blk_rowxfer (blkdesc) == CS_FAIL)
           {
               ex_error("BulkCopyIn: blk_rowxfer() failed");
               return CS_FAIL;
           }
           dptr++;
       }

       /* Mark the operation complete and then clean up */
       ...CODE DELETED.....

       return CS_SUCCEED;
   }
```

Usage

- blk_bind is a client-side routine.

- blk_bind binds program variables to table columns in the database. Once variables are bound, subsequent calls to blk_rowxfer_mult copy row data between the database and the bound variables. The copy direction is determined by the application's earlier call to blk_init.
- When copying into a database, an application must call blk_bind once for each column in the database table. When copying out, an application need not call blk_bind for columns in which it has no interest.
- To indicate that a column value will be transferred using blk_textxfer, an application calls blk_bind with *buffer* as NULL. A typical application will use blk_textxfer to transfer large text or image values.

  If a text, image, boundary, or sensitivity datatype column is marked for transfer using blk_textxfer, all subsequent columns of these types must also be marked for transfer using blk_textxfer. For example, an application cannot mark the first text column in a row for transfer using blk_textxfer and then bind a subsequent text column to a program variable.
- An application can call blk_bind in between calls to blk_rowxfer_mult to reflect changes in a variable's address or length. If an application calls blk_bind multiple times for a single column or variable, only the last binding takes effect.
- An application can call blk_describe to initialize a CS_DATAFMT structure that describes the format of a particular column.

blk_bind for bulk-copy-in operations

Table 4-2 summarizes blk_bind usage when used for bulk-copy-in operations. For information on *datafmt* fields, see Table 4-1 on page 122.

Table 4-2: blk_bind parameter values for bulk copy in

When calling blk_bind to	buffer is	datalen is	*indicator is
Bind to a scalar or array variable from which blk_rowxfer_mult will read column values	The address of a program variable or array	A pointer to a variable or array that indicates the length of the values to be read from **buffer*. • If **datalen* is greater than 0, **datalen* values are read from **buffer* and sent as the column value. • When **datalen* is 0, the value of **indicator* is used to determine whether the column's default value (if any) or NULL should be inserted.	The address of a variable or array that supplies indicator values for the column. **indicator* is only considered when **datalen* is 0: • If **indicator* is 0, the column's default value (if available) is inserted. If no default value is available, a NULL is inserted. • If **indicator* is -1, NULL is always inserted.
Indicate that a column value will be transferred using blk_textxfer	NULL	The total length of the data that will be sent using blk_textxfer. In this case, *datafmt->maxlength* is ignored.	Ignored.

When a Bulk-Library application calls blk_bind in a bulk-copy-in operation the *buffer*, *datalen*, and *indicator* pointers passed to blk_bind are recorded. The data at those locations must remain valid until it is read during the call to blk_rowxfer or blk_rowxfer_mult.

blk_bind for Bulk-Copy-Out operations

Table 4-3 summarizes blk_bind usage when used for bulk-copy-out operations. For information on *datafmt* fields, see Table 4-1 on page 122.

Table 4-3: blk_bind parameter values for bulk copy out

When calling blk_bind to	buffer is	*datalen is	*indicator is
Bind to a scalar or array variable into which blk_rowxfer_mult will write column values	The address of a program variable or array	A pointer to a variable or to a CS_INT variable for an array, where blk_rowxfer_mult on page 160 places the length of the values written to **buffer*.	The address of a variable or array that supplies indicator values for the column. blk_rowxfer_mult sets **indicator* as follows: • -1 indicates the data is null. • 0 indicates good data. • A value greater than 0 indicates truncation occurred. The value is the actual length of the available data.
Indicate that a column value will be transferred using blk_textxfer	NULL	Ignored. In this case, *datafmt–>maxlength* represents the length of the **buffer* data space.	Ignored.

Specifying Null values for Bulk Copy into the database

- When copying in, an application can instruct blk_rowxfer_mult to use a column's default value by setting **datalen* to 0 and **indicator* to 0 before calling blk_rowxfer_mult. If no default value is defined for the column, blk_rowxfer_mult inserts a NULL value.
- To instruct blk_rowxfer_mult to insert a NULL regardless of a column's default value, set **datalen* to 0 and **indicator* to -1 before calling blk_rowxfer_mult.

Clearing bindings

- To clear a binding, call blk_bind with *buffer*, *datafmt*, *datalen*, and *indicator* as NULL. Otherwise, bindings remain in effect until an application calls blk_done with *type* as CS_BLK_ALL to indicate that the bulk-copy operation is complete.
- To clear all bindings, pass *colnum* as CS_UNUSED, with *buffer*, *datafmt*, *datalen*, and *indicator* as NULL. An application typically clears all bindings when it needs to change the count that is being used for array binding.

Array binding

- Array binding is the process of binding a column to an array of program variables. At row-transfer time, multiple rows of the column are transferred either to or from the array of variables with a single blk_rowxfer_mult call. An application indicates array binding by setting *datafmt–>count* to a value greater than 1.
- Array binding works differently for bulk-copy-in and bulk-copy-out operations.
- For bulk-copy-in operations that use array binding, you must call blk_bind with *buffer*, *datalen*, and *indicator* pointing to arrays. Each length and indicator variable describes the corresponding data in the buffer array. For fixed-length data, *buffer* is always a pointer to an array of fixed-length values. For variable-length data (specifically character or binary data), *buffer* is a pointer to an array of bytes. In the latter case, the packing of values can be *loose* or *dense*. The application specifies the packing method for each column by setting flags in the *datafmt–>format* field:
  - Setting the CS_BLK_ARRAY_MAXLEN bit in *datafmt–>format* specifies *loose* packing of values in the array. blk_rowxfer_mult retrieves the value *i* by reading *datalen*[*i*-1] bytes starting at the byte position computed as:

    ```
    (i -1) * datafmt–>maxlength
    ```

  - If the CS_BLK_ARRAY_MAXLEN bit is not set in *datafmt–>format*, column values must be densely packed for blk_rowxfer_mult. Each value must be placed in the column array immediately after the previous value, without padding. blk_rowxfer_mult gets value *i* by reading *datalen*[*i*-1] bytes starting at the byte position computed as:

    ```
    datalen[i-2] + datalen[i-3] + ... + datalen[0]
    ```

    In other words, the first value starts at 0, the second at *datalen*[0], the third at *datalen*[1] + *datalen*[0], and so forth.

  For example, consider a character column that will receive the values “girl,” “boy,” “man,” and “woman,” and assume that this column is bound with *datafmt–>maxlength* passed as 7. With loose array binding, the *buffer* and *datalen* contents would be:

  ```
  buffer: girl   boy    man    woman
            0      7      14     21
   datalen: 4, 3, 3, 5
  ```

  With densely-packed array binding, the *buffer* and *datalen* contents would be:

```
buffer: girlboymanwoman
          0   4  7  10
 datalen: 4, 3, 3, 5
```

- For bulk-copy-out operations, array binding performed with blk_bind works the same as array binding performed with ct_bind. Column arrays for bulk-copy-out are always loosely packed.
- While using array binding during a bulk-copy-out operation, it is possible for conversion, memory, or truncation errors to occur while blk_rowxfer_mult is writing to the destination arrays. In this case, blk_rowxfer_mult writes a partial result to the destination arrays and returns CS_ROW_FAIL.
- If array binding is in effect (for either direction), an application cannot use blk_textxfer to transfer data.

See also

blk_describe, blk_default, blk_init

blk_colval

Description

A server-side routine, obtains the column value from a formatted bulk-copy row.

Syntax

```
CS_RETCODE blk_colval(srvproc, blkdescp, rowp, colnum,
        valuep, valuelen, outlenp)

SRV_PROC      *srvproc;
CS_BLKDESC    *blkdescp;
CS_BLK_ROW    *rowp;
CS_INT         colnum;
CS_VOID        *valuep;
CS_INT        valuelen;
CS_INT        *outlen;
```

Parameters

srvproc

A pointer to the SRV_PROC structure associated with the client sending the bulk-copy row. It contains all the information that Server-Library uses to manage communications and data between the Open Server application and the client.

blkdescp
A pointer to a CS_BLKDESC structure containing information about bulk-copy data. This structure must have been previously allocated with a call to blk_alloc and initialized with a call to blk_init. This structure is used to interpret incoming formatted bulk-copy rows.

rowp
A pointer to the CS_BLK_ROW structure filled in by a prior call to blk_getrow.

The CS_BLK_ROW structure is a hidden structure that holds formatted bulk-copy rows sent from the client.

colnum
The column number of the column of interest. Column numbers start at 1.

valuep
A pointer to the application buffer in which the column value from the bulk-copy row is placed.

valuelen
The size, in bytes, of the buffer to which *valuep* points.

outlen
A pointer to a CS_INT variable. blk_colval sets **outlen* to the size, in bytes, of the column data.

Return value

blk_colval returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- blk_colval is a server-side routine. After getting the value of a specified column from a formatted bulk-copy row, it stores the value in an application buffer.
- This routine performs no implicit data conversion. Use cs_convert to convert the data.
- To examine the column value after a call to blk_colval, the application must know the column's datatype before making the call.
- An Open Server application cannot use this routine to retrieve text, image, sensitivity, or boundary columns. Use blk_gettext to retrieve such columns.

See also

blk_getrow, blk_gettext

blk_default

Description

Retrieves a column's default value.

Syntax

```
CS_RETCODE blk_default(blkdesc, colnum, buffer,
        buflen, outlen)
CS_BLKDESC   *blkdesc;
CS_INT        colnum;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT       *outlen;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that serves as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

colnum

The number of the column of interest. The first column in a table is column number 1, the second is number 2, and so forth.

buffer

A pointer to the space in which blk_default will place the default value.

buflen

The length, in bytes, of the **buffer* data space.

outlen

A pointer to an integer variable.

If supplied, blk_default sets **outlen* to the length, in bytes, of the default value.

If the default value is larger than *buflen* bytes, an application can use the value of **outlen* to determine how many bytes are needed to hold the value.

Return value

blk_default returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

blk_default returns CS_FAIL if the application has not called blk_init to initialize the bulk-copy operation.

Usage

- blk_default is a client-side routine.
- An application can call blk_default to find out whether a default value is defined for a particular target column, and, if so, what the default value is.

This information can be useful while preparing to bulk copy rows into a database. The application can set **datalen* and **indicator* values to specify whether a column's default value should be used. (*datalen* and *indicator* are the addresses of program variables that were bound to the column with blk_bind). See "Specifying Null values for Bulk Copy into the database" on page 130.

- If the column of interest does not have a default value, blk_default sets **outlen* to CS_NO_DEFAULT and returns CS_SUCCEED.
- An application can retrieve column defaults with blk_default only during a bulk-copy-in operation. The application cannot call blk_default until blk_init(CS_BLK_IN) returns CS_SUCCEED.

See also

blk_bind, blk_describe, blk_init

blk_describe

Description

Retrieves a description of a database column.

Syntax

CS_RETCODE blk_describe(blkdesc, colnum, datafmt)

```
CS_BLKDESC      *blkdesc;
CS_INT          colnum;
CS_DATAFMT      *datafmt;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

colnum

The number of the column of interest. The first column in a table is column number 1, the second is number 2, and so forth.

datafmt

A pointer to a CS_DATAFMT structure. blk_describe fills **datafmt* with a description of the database column referenced by *colnum*.

During a bulk-copy-in operation, blk_describe fills in the following fields in the CS_DATAFMT:

Table 4-4: CS_DATAFMT fields, as set by blk_describe for bulk-copy-in

Field name	blk_describe sets the field to
name	The null-terminated name of the column, if any. A NULL name is indicated by a *namelen* of 0.
namelen	The actual length of the name, not including the null terminator. 0 indicates a NULL *name*.
datatype	A type constant representing the datatype of the column. All type constants listed on the "Types" topics page are valid, with the exception of CS_VARCHAR_TYPE and CS_VARBINARY_TYPE.
maxlength	The maximum possible length of the data for the column.
scale	The scale of the column.
precision	The precision of the column.

During a bulk-copy-out operation, blk_describe fills in the following fields in the CS_DATAFMT:

Table 4-5: CS_DATAFMT fields, as set by blk_describe for bulk-copy-out

Field name	blk_describe sets the field to
name	The null-terminated name of the column, if any. A NULL name is indicated by a *namelen* of 0.
namelen	The actual length of the name, not including the null terminator. 0 indicates a NULL *name*.
datatype	The datatype of the column. All datatypes listed on the "Types" topics page in the *Open Client Client-Library/C Reference Manual* are valid.
maxlength	The maximum possible length of the data for the column.
scale	The scale of the column.
precision	The precision of the column.
status	A bit mask of the following symbols, combined with a bitwise, OR: • CS_CANBENULL to indicate that the column can contain NULL values. • CS_HIDDEN to indicate that this column is a hidden column that has been exposed. Hidden columns are exposed when the CS_HIDDEN_KEYS property is set for the bulk descriptor's parent connection. • CS_IDENTITY to indicate that the column is an identity column. • CS_KEY to indicate the column is part of the key for a table. • CS_VERSION_KEY to indicate the column is part of the version key for the row.
usertype	The Adaptive Server Enterprise user-defined datatype of the column, if any. *usertype* is set in addition to (not instead of) *datatype*.
locale	A pointer to a CS_LOCALE structure that contains locale information for the data.

Return value

blk_describe returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

blk_describe returns CS_FAIL if *colnum* does not represent a valid result column.

Usage

- blk_describe is a client-side routine.
- blk_describe describes the format of a database column. The application can use this information to:

- Determine the datatype and size requirements for allocating storage for retrieving rows (for bulk copy out of the database).
- Determine compatibility between program variable datatypes and the database columns (by calling cs_will_convert to determine whether the conversion is supported and, if necessary, by checking the data lengths).
- Perform error checking. For example, the debug version of a bulk-copy application might call blk_describe to confirm assumptions about the format of table columns.

- An application typically uses a column description while determining compatible program variable types and sizes.
- See the "CS_DATAFMT Structure" topics page in the *Open Client Client-Library/C Reference Manual* for a complete description of the CS_DATAFMT structure.

See also

blk_default, blk_init

blk_done

Description

Marks a complete bulk-copy operation or a complete bulk-copy batch.

Syntax

```
CS_RETCODE blk_done(blkdesc, type, outrow)

CS_BLKDESC      *blkdesc;
CS_INT           type;
CS_INT          *outrow;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

type

One of the following symbolic values:

Value of type	blk_done
CS_BLK_ALL	Marks a complete bulk-copy-in or bulk-copy-out operation.
CS_BLK_BATCH	Marks the end of a batch of rows in a batched bulk-copy-in operation.
CS_BLK_CANCEL	Cancels a bulk-copy batch or bulk-copy operation.

outrow

A pointer to an integer variable. If *type* is CS_BLK_BATCH or CS_BLK_ALL, blk_done sets **outrow* to the number of rows bulk copied to Adaptive Server Enterprise since the application's last blk_done call. *W*hen *type* is CS_BLK_CANCEL, **outrow* is set to 0.

Return value

blk_done returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.

Common reasons for blk_done failure include:

- An invalid *blkdesc* pointer
- An invalid value for *type*

Examples

```
/*
 ** BulkCopyIn()
 */


CS_STATIC CS_RETCODE
 BulkCopyIn(connection)
 CS_CONNECTION   *connection;
 {
      CS_BLKDESC   *blkdesc;
      CS_DATAFMT   datafmt;     /* variable descriptions */
      Blk_Data     *dptr;       /* data for transfer */
      CS_INT       datalen[5];  /* variable data length */
      CS_INT       len;
      CS_INT       numrows;

      /*
      ** Ready to start the bulk copy in now that all the
      ** connections have been made and have a table name.
      ** Start by getting the bulk descriptor initializing.
      */
      ...CODE DELETED.....

      /*
```

```
        ** Now to bind the variables to the columns and

        ** transfer the data
        */
        ...CODE DELETED.....

        /* ALL the rows sent so clear up */
        if (blk_done(blkdesc, CS_BLK_ALL, &numrows) == CS_FAIL)
        {
           ex_error("BulkCopyIn: blk_done() failed");
           return CS_FAIL;
        }
        if (blk_drop(blkdesc) == CS_FAIL)
        {
           ex_error("BulkCopyIn: blk_drop() failed");
           return CS_FAIL;
        }
         return CS_SUCCEED;
    }
```

Usage

- A client-side routine called blk_done is necessary in both client-only and gateway applications.

  Note Setting CS_OPT_NOCOUNT before doing a bulk copy operation on a connection, causes blk_done to erroneously report errors.

- Calling blk_done with *type* as CS_BLK_ALL marks the end of a bulk-copy operation. Once an application marks the end of a bulk-copy operation, it cannot call any Bulk-Library routines (except for *blk_drop* and *blk_alloc*) until it begins a new bulk-copy operation by calling blk_init.
- Calling blk_done with *type* as CS_BLK_BATCH marks the end of a batch of rows in a bulk-copy-in operation. CS_BLK_BATCH is legal only during bulk-copy-in operations.
- Calling blk_done with *type* as CS_BLK_CANCEL cancels the current bulk-copy operation. Rows transferred since an application's last blk_done(CS_BLK_BATCH) call are not saved in the database. Once an application cancels a bulk-copy operation, it cannot call any bulk-copy routines (except for blk_drop and blk_alloc) until it initializes a new bulk-copy operation by calling blk_init.

Calling *blk_done* during Bulk-Copy-In operations

- When an application bulk copies data into a database, the rows are permanently saved only when the application calls blk_done. During a large data transfer, blk_done(CS_BLK_BATCH) can be called periodically to "batch" the transmitted rows into smaller units of recoverability.
- An application can batch rows by calling blk_done with *type* as CS_BLK_BATCH once every *n* rows or when there is a lull between periods of data, as in a telemetry application. This causes all rows transferred since the application's last blk_done call to be permanently saved.
- After saving a batch of rows, an application's first call to blk_rowxfer or blk_rowxfer_mult implicitly starts the next batch.
- An application must call blk_done with type as CS_BLK_ALL to send its final batch of rows. This call permanently saves the rows, marks the end of the bulk-copy operation, and cleans up internal bulk-copy data structures.

Calling *blk_done* during bulk-copy-out operations

- After transferring the last row in a bulk-copy-out operation, an application must call blk_done with type as CS_BLK_ALL to mark the end of the bulk-copy operation and clean up internal bulk-copy data structures.

See also

blk_init, blk_rowxfer, blk_rowxfer_mult

blk_drop

Description

Deallocates a CS_BLKDESC structure.

Syntax

```
CS_RETCODE blk_drop(blkdesc)

 CS_BLKDESC      *blkdesc;
```

Parameters

blkdesc

A pointer to a CS_BLKDESC previously allocated through blk_alloc.

Return value

blk_drop returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
/*
 ** BulkCopyIn()
 */

 CS_STATIC CS_RETCODE
 BulkCopyIn(connection)
 CS_CONNECTION   *connection;
  {

 CS_BLKDESC   *blkdesc;
 CS_DATAFMT   datafmt;      /* variable descriptions */
 Blk_Data     *dptr;        /* data for transfer */
 CS_INT       datalen[5];   /* variable data length */
 CS_INT       len;

 CS_INT       numrows;

 /*
 ** Ready to start the bulk copy in now that all the
 ** connections have been made and have a table name.
 ** Start by getting the bulk descriptor initializing.
 */
 ...CODE DELETED.....

 /*
 ** Now to bind the variables to the columns and
 ** transfer the data
 */
 ...CODE DELETED.....

 /* ALL the rows sent so clear up */
 if (blk_done(blkdesc, CS_BLK_ALL, &numrows) == CS_FAIL)
 {
      ex_error("BulkCopyIn: blk_done() failed");
      return CS_FAIL;
 }
 if (blk_drop(blkdesc) == CS_FAIL)
 {
      ex_error("BulkCopyIn: blk_drop() failed");
      return CS_FAIL;
 }

 return CS_SUCCEED;
 }
```

Usage

- A CS_BLKDESC structure, also called a *bulk-descriptor structure*, contains information about a particular bulk-copy operation.
- Once a bulk-descriptor structure has been deallocated, it cannot be used again. To allocate a new CS_BLKDESC, an application can call blk_alloc.
- blk_drop is typically called after blk_done. It must be the last routine called in a bulk-copy operation.

See also

blk_alloc, blk_done

blk_getrow

Description

Server-side routine retrieves and stores a formatted bulk-copy row.

Syntax

```
CS_RETCODE blk_getrow(srvproc, blkdescp, rowp)

  SRV_PROC      *srvproc;
CS_BLKDESC    *blkdescp;
CS_BLK_ROW     *rowp;
```

Parameters

srvproc

A pointer to the SRV_PROC structure associated with the client sending the bulk-copy row. It contains all the information that Server-Library uses to manage communications and data between the Open Server and the client.

blkdescp

A pointer to a CS_BLKDESC structure containing information about bulk-copy data. This structure must have been previously allocated with a call to blk_alloc and initialized with a call to blk_init. This structure is used to interpret incoming formatted bulk-copy rows.

rowp

A pointer to a CS_BLK_ROW structure containing space for a formatted bulk-copy row. Space must have been previously allocated with blk_rowalloc.

The CS_BLK_ROW structure is a hidden structure that holds formatted bulk-copy rows sent from the client.

Return value

blk_getrow returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.

Returns	Indicates
CS_END_DATA	There are no more rows.
CS_BLK_HAS_TEXT	The row contains some text, image, sensitivity, or boundary data. Use blk_gettext to retrieve the text, image, sensitivity, or boundary data. Note that a return value of CS_BLK_HAS_TEXT implies a successful return, just like CS_SUCCEED.
CS_FAIL	The routine failed.

Usage

- blk_getrow is a server-side routine that is useful in gateway applications.
- This routine copies the incoming formatted bulk-copy row into the CS_BLK_ROW structure to which *rowp* points. The row data is saved only until the next call to blk_getrow. The application must have previously allocated the space for the row using blk_rowalloc.
- Once a row has been received through blk_getrow, the application may examine the contents of any fields (other than text, image, sensitivity, or boundary fields) using blk_colval.
- Use blk_gettext to retrieve text, image, sensitivity, and boundary fields.
- A bulk-copy row may subsequently be sent to another server using the blk_sendrow routine.
- An application must read all incoming rows with blk_getrow, until there are no more rows.
- Once blk_getrow returns CS_END_DATA, the application must drop the space allocated for the row using blk_rowdrop.

See also

blk_colval, blk_gettext, blk_rowalloc

blk_gettext

Description

Server-side routine retrieves the text, image, sensitivity, or boundary portion of an incoming formatted bulk-copy row.

Syntax

CS_RETCODE blk_gettext(srvproc,blkdescp, rowp, bufp, bufsize, outlenp)

```
  SRV_PROC      *srvproc;
CS_BLKDESC    *blkdescp;
CS_BLK_ROW     *rowp;
CS_BYTE         *bufp;
```

```
CS_INT          bufsize;
CS_INT          *outlenp;
```

Parameters

srvproc

A pointer to the SRV_PROC structure associated with the client sending the bulk-copy row. This structure contains all the information that Server-Library uses to manage communications and data between the Open Server application and the client.

blkdescp

A pointer to a CS_BLKDESC structure containing information about bulk-copy data. This structure must have been previously allocated with a call to blk_alloc and initialized with a call to blk_init. This structure is used to interpret incoming formatted bulk-copy rows.

rowp

A pointer to the formatted bulk-copy row read from the client through a prior call to blk_getrow.

The CS_BLK_ROW structure is a hidden structure that holds formatted bulk-copy rows sent from the client.

bufp

A pointer to the application buffer in which Bulk-Library places the text, image, sensitivity, or boundary data.

bufsize

The size, in bytes, of the space pointed at by *bufp*.

outlenp

A pointer to a CS_INT variable, which is set to the number of bytes actually read by blk_gettext. It may be less than *bufsize*. To determine whether all of the text, image, sensitivity, or boundary part of the row has been read, check for a return code of CS_END_DATA. An **outlenp* value that is less than *bufsize* does not necessarily indicate the end of a row. For example, it could indicate the end of a text, image, sensitivity, or boundary column that is not the last column in the row.

Return value

blk_gettext returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_END_DATA	There are no more text, image, sensitivity, or boundary fields for the current incoming bulk-copy row. Call blk_getrow to get the next bulk-copy row.
CS_FAIL	The routine failed.

Usage

- blk_gettext is a server-side routine that is useful in gateway applications.
- This routine is used with blk_getrow and blk_colval to receive formatted bulk-copy rows and route them to an Adaptive Server Enterprise. This routine retrieves the text, image, sensitivity, or boundary portions of the row.
- Bulk-copy rows are formatted so that all text, image, sensitivity, and boundary fields occur at the end of the row, after all the other types of fields. To route a row to an Adaptive Server Enterprise, first call blk_getrow to retrieve all the parts of the row containing other types of fields. Then, call blk_colval to retrieve and store portions of the row containing other types of fields. Decide where this data goes and send it to the remote server, using blk_sendrow. Call blk_gettext to copy text, image, sensitivity, or boundary data into an application buffer. Finally, call blk_sendtext to send this information to the remote server.
- If an incoming bulk-copy row has any text, image, sensitivity, or boundary fields, blk_getrow returns CS_BLK_HAS_TEXT.
- It is not an error to call blk_gettext if the row contains no text, image, sensitivity, or boundary fields. The routine simply returns CS_END_DATA.
- This routine must be called after blk_getrow. Also, it must be called until it returns CS_END_DATA, to fully read in a bulk-copy row.
- Before rows can be sent to a server, the gateway application must have set up the bulk-copy operation with a call to blk_init.
- It is critical that the table for which the bulk-copy operation was initialized and the table into which the client is bulk copying are the same table.

See also

blk_colval, blk_getrow, blk_gettext, blk_sendtext

blk_init

Description

Initiates a bulk-copy operation.

Syntax

```
CS_RETCODE blk_init(blkdesc, direction, tablename,
        tnamelen)

CS_BLKDESC    *blkdesc;
CS_INT        direction;
```

```
CS_CHAR         *tablename;
CS_INT          tnamelen;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC controlling the bulk-copy operation. An application can allocate a CS_BLKDESC by calling blk_alloc.

The parent connection of the CS_BLKDESC must be open when blk_init is called and cannot have any pending results.

direction

One of the following symbolic values, to indicate the direction of the bulk-copy operation:

Value of direction	blk_init
CS_BLK_IN	Begins a bulk-copy operation to upload rows from the client to the server.
CS_BLK_OUT	Begins a bulk-copy operation to download rows from the server to the client.

tablename

A pointer to the name of the table of interest. Any legal server table name is acceptable. The table name cannot contain a colon (:) or slice number.

tnamelen

The length, in bytes, of **tablename*. If **tablename* is null-terminated, pass *tnamelen* as CS_NULLTERM.

Return value

blk_init returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.

A common cause of failure is specifying a non-existent table.

Examples

```
/*
 ** BulkCopyIn()
 ** Ex_tabname is globally defined.
 */
 CS_STATIC CS_RETCODE
 BulkCopyIn(connection)
```

```
CS_CONNECTION   *connection;
{
    CS_BLKDESC *blkdesc;
    CS_DATAFMT datafmt;   /* variable descriptions */
    Blk_Data *dptr;      /* data for transfer */
    CS_INT datalen[5];    /* variable data length */
    CS_INT len;
    CS_INT numrows;

    /*
    ** Ready to start the bulk copy in now that all the
    ** connections have been made and have a table name.
    ** Start by getting the bulk descriptor and
    ** initializing.
    */
    if (blk_alloc(connection, BLK_VERSION_100, &blkdesc)
        != CS_SUCCEED)
    {
        ex_error("BulkCopyIn: blk_alloc() failed");
        return CS_FAIL;
    }

    if (blk_init(blkdesc, CS_BLK_IN,
                Ex_tabname, strlen(Ex_tabname)) == CS_FAIL)
    {
        ex_error("BulkCopyIn: blk_init() failed");
        return CS_FAIL;
    }

    /*
    ** Bind the variables to the columns and send the rows,
    ** and then clean up.
    */
    ...CODE DELETED.....

    return CS_SUCCEED;
}
```

Usage

- blk_init begins a bulk-copy operation.
- blk_init is a client-side routine. However, it is necessary in both client-only and gateway applications.
- Multiple CS_BLKDESC and CS_COMMAND structures can exist on the same connection, but only one CS_BLKDESC or CS_COMMAND structure can be active at the same time.

- A bulk-copy operation begun with blk_init must be completed before the connection can be used for any other operation.
- A bulk-copy operation cannot be started when the connection is being used to initiate, send, or process the results of other Client-Library or Bulk-Library commands.

- When a bulk-copy operation is complete, an application must call blk_done with *type* as CS_BLK_ALL to mark the end of the bulk-copy operation and clean up internal Bulk-Library data structures.

See also

blk_alloc, blk_bind, blk_done, blk_rowxfer_mult

blk_props

Description

Sets or retrieves bulk-descriptor structure properties.

Syntax

```
CS_RETCODE blk_props(blkdesc, action, property,
        buffer, buflen, outlen)

CS_BLKDESC      *blkdesc;
CS_INT          action;
CS_INT          property;
CS_VOID         *buffer;
CS_INT          buflen;
CS_INT          *outlen;
```

Parameters

blkdesc

A pointer to a CS_BLKDESC structure. A bulk-descriptor structure contains information about a bulk-copy operation. blk_alloc allocates a bulk-descriptor structure.

action

One of the following symbolic constants:

Value of action	blk_props
CS_SET	Sets the value of the property
CS_GET	Retrieves the value of the property
CS_CLEAR	Clears the value of the property by resetting it to its default value

property

A symbolic constant that indicates the property of interest. Table 4-6 on page 151 lists valid *property* constants and describes each property.

buffer

If a property value is being set, *buffer* points to the value to use in setting the property.

If a property value is being retrieved, *buffer* points to the space in which blk_props will place the requested information.

The C datatype of the value depends on the property. Refer to Table 4-6 on page 151 for the datatype of the property of interest.

buflen

Generally, *buflen* is the length, in bytes, of **buffer*.

If a property value is being set and the value in **buffer* is null-terminated, pass *buflen* as CS_NULLTERM.

If **buffer* is a fixed-length or symbolic value, pass *buflen* as CS_UNUSED.

outlen

A pointer to an integer variable.

If a property value is being set, *outlen* is not used and should be passed as NULL.

If a property value is being retrieved and *outlen* is supplied, blk_props sets **outlen* to the length, in bytes, of the requested information.

If the information is larger than *buflen* bytes, an application can use the value of **outlen* to determine how many bytes are needed to hold the information.

Return value

blk_props returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- Bulk-descriptor properties define aspects of a specific bulk-copy operation.
- Applications that set Bulk-Library properties must do so after calling blk_alloc to allocate a bulk-descriptor structure and before calling blk_init to initiate a specific bulk-copy operation.
- An application can use blk_props to set or retrieve the following properties:

Table 4-6: Client/Server bulk descriptor properties

Property name	Description	*buffer is	Applies to	Notes
BLK_CONV	Character-set conversion performed by client.	CS_TRUE or CS_FALSE.	IN copies only	To disable character-set conversion on the current server connection, set CS_NOCHARSETCNV_REQD to CS_TRUE. See the ct_con_props section in the *Open Client Client-Library/C Reference Manual*.
BLK_IDENTITY	Whether values for a table's identity column are specified explicitly for each row to be inserted. This property cannot be set to CS_TRUE if BLK_IDSTARTNUM has been set for a bulk-copy-in operation.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which indicates that identity values are either: • Computed from the starting value indicated by BLK_IDSTARTNUM, or • Computed by Adaptive Server Enterprise as data is inserted, based on existing identity values in the table.	IN copies only	

Property name	Description	*buffer is	Applies to	Notes
BLK_IDSTARTNUM	The starting value for identity columns in inserted rows. The first inserted row uses this value, and the value is incremented for each subsequent row. This property cannot be set if BLK_IDENTITY has been set to CS_TRUE for the bulk-copy-in operation.	A CS_NUMERIC variable containing the starting identity value. There is no default.	IN copies only	
BLK_NOAPI_CHK	Whether parameter and error checking for illegal parameter values and state transitions are disabled for Bulk-Library calls.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means error checking is performed.	Both IN and OUT copies	
BLK_PARTITION	Property to support BCP partitions for BCP_IN and BCP_OUT operations.	A character string containing the name of the partition.	Both IN and OUT copies	
BLK_SENSITIVITY_ LBL	Whether a table's *sensitivity* column is included in the bulk-copy operation.	CS_TRUE or CS_FALSE (default).	Both IN and OUT copies	Secure Adaptive Server Enterprise only
BLK_SLICENUM	For bulk-copy into a partitioned table. Specifies the partition number that copied rows are inserted to.	A CS_INT variable containing a positive value representing the partition number. The default is CS_UNUSED, which indicates that Adaptive Server Enterprise will randomly choose a partition number.	IN copies only	

BLK_IDENTITY property

- BLK_IDENTITY determines whether a table's identity column is included in a bulk-copy-in operation.
- BLK_IDENTITY does not affect bulk-copy-out operations.
- If BLK_IDENTITY is CS_TRUE, the application must supply data for the identity column.

  If BLK_IDENTITY is CS_FALSE, the application does not need to supply data for the identity column. In this case, the server supplies a default value for the column.
- BLK_IDENTITY works by setting identity_insert on for the database table of interest. This allows values to be inserted into the identity column. When the bulk-copy operation is finished, the identity_insert option for the table is turned off.

  See the *Adaptive Server Enterprise Reference Manual*.

BLK_NOAPI_CHK property

- BLK_NOAPI_CHK can be set to CS_TRUE to disable parameter and state checking of Bulk-Library calls. The default is CS_FALSE, which enables parameter checking and state checking of each Bulk-Library call. These two types of error checking are described below:
  - *Parameter checking* determines whether the application has passed valid parameters and combinations of parameters in the call.
  - *State checking* ensures that calls are made in the required sequence. For example, blk_init must be called before blk_bind.

  The default error checking ensures that your application calls Bulk-Library routines in the appropriate manner. With API checking enabled, a descriptive error message is raised when the application commits a usage error, and the routine that discovers the error returns CS_FAIL.

  Warning! With API checking disabled, Bulk-Library usage errors may lead to unexpected behavior or even program crashes.

- If your application has been fully tested and completely debugged, you may see improved performance with API checking disabled. Bulk-Library also calls Client-Library internally, so to get the full benefit, you should also disable API checking in Client-Library (by calling ct_config to set the CS_NOAPI_CHK context property to CS_TRUE).

- BLK_NOAPI_CHK does not affect testing for errors, such as network errors or conversion overflow, that can occur in well-behaved applications.

BLK_SENSITIVITY_LBL property

- BLK_SENSITIVITY_LBL is useful in applications that perform bulk-copy operations to or from Secure Adaptive Server Enterprise.
- BLK_SENSITIVITY_LBL determines whether or not data for the *sensitivity* column is included in a bulk-copy operation. By default, *sensitivity* column data is not included.
- BLK_SENSITIVITY_LBL affects both bulk-copy-in and bulk-copy-out operations.
- If BLK_SENSITIVITY_LBL is CS_TRUE, the application must supply data for the *sensitivity* column on bulk-copy-in operations and will receive data from the *sensitivity* column on bulk-copy-out operations.

  If BLK_SENSITIVITY_LBL is CS_FALSE, the application does not need to supply data for the *sensitivity* column on bulk-copy- in operations and will not receive data from the *sensitivity* column on bulk-copy-out operations.
- BLK_SENSITIVITY_LBL is applicable to Secure Adaptive Server Enterprise copies only. blk_init fails if BLK_SENSITIVITY_LBL is CS_TRUE and the application attempts a bulk-copy operation against a standard Adaptive Server Enterprise.
- Application users copying into the *sensitivity* column must have the bcpin_labels_role role activated on Secure Adaptive Server Enterprise. blk_init fails if the bcpin_labels_role is not activated for the connection's user.
- See your Secure Adaptive Server Enterprise documentation.

BLK_PARTITION property

- Only one name can be provided. A single BLKLIB operation always operates on an entire table or on a single partition. If no partition name is provided, the BLKLIB will not operate on a specific partition but on the entire table.
- This property can be used for both BCP_IN and BCP_OUT operations. Either BLK_PARTITION or BLK_SLICENUM can be used; if one is set, the other is cleared.

- The BLK_PARTITION property does not require you to set CS_VERSION_155 or BLK_VERSION_155.

See also blk_alloc, blk_init

blk_rowalloc

Description A server-side routine, allocates space for a formatted bulk-copy row.

Syntax

```
CS_RETCODE blk_rowalloc(srvproc, row)

SRV_PROC       *srvproc;
CS_BLK_ROW     **row;
```

Parameters

srvproc

A pointer to the SRV_PROC structure associated with the client sending formatted bulk-copy rows. It contains all the information that Server-Library uses to manage communications and data between the Open Server and the client.

row

A pointer to a pointer to a CS_BLK_ROW structure.

The CS_BLK_ROW structure is a hidden structure that holds formatted bulk-copy rows sent from the client.

Return value blk_rowalloc returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- blk_rowalloc is a server-side routine that is useful in gateway applications.
- This routine allocates space in which blk_getrow will place the formatted bulk-copy row.
- The row space is used by all calls to blk_getrow.
- When all rows have been retrieved and sent to the remote server, call blk_rowdrop to drop the space allocated for the row.

See also blk_getrow, blk_rowdrop, blk_gettext

blk_rowdrop

Description

A server-side routine, frees space previously allocated for a formatted bulk-copy row.

Syntax

```
CS_RETCODE blk_rowdrop(srvproc, row)

SRV_PROC        *srvproc;
CS_BLK_ROW      *row;
```

Parameters

srvproc

A pointer to the SRV_PROC structure associated with the client sending formatted bulk-copy rows. It contains all the information that Server-Library uses to manage communications and data between the Open Server application and the client.

row

A pointer to a hidden CS_BLK_ROW structure that was allocated by a call to blk_rowalloc.

Return value

blk_rowdrop returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- blk_rowdrop is a server-side routine that is useful in gateway applications.
- This routine frees space previously allocated by blk_rowalloc.
- It must be called after all formatted bulk-copy rows have been retrieved and sent to the remote server.

See also

blk_getrow, blk_rowalloc, blk_gettext

blk_rowxfer

Description

Transfers one or more rows during a bulk-copy operation without specifying or receiving a row count.

Syntax

```
CS_RETCODE blk_rowxfer(blkdesc)

CS_BLKDESC      *blkdesc;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

Return value

blk_rowxfer returns:

Table 4-7: blk_rowxfer return values

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.
CS_BLK_HAS_TEXT	The row contains one or more columns which have been marked for transfer using blk_textxfer. The application must call blk_textxfer to transfer data for these columns before calling blk_rowxfer to transfer the next row.
CS_END_DATA	When copying data out from a database, blk_rowxfer returns CS_END_DATA to indicate that all rows have been transferred. When copying data into a database, blk_rowxfer does not return CS_END_DATA.
CS_ROW_FAIL	A recoverable error occurred while fetching a row. Applies to bulk-copy-out operations only. Recoverable errors include memory allocation failures and conversion errors (such as overflowing the destination buffer) that occur while copying row values to program variables. In the case of buffer-overflow errors, blk_rowxfer sets the corresponding **indicator* variable(s) to a value greater than 0. Indicator variables must have been specified in the application's calls to blk_bind. When blk_rowxfer returns CS_ROW_FAIL, the application must continue calling blk_rowxfer to keep retrieving rows, or it can call ct_cancel to cancel the remaining results.

Examples

```
/*
 ** BulkCopyIn()
 ** BLKDATA and DATA_END are defined in the bulk copy
 ** example program.
 */
```

```
CS_STATIC CS_RETCODE
BulkCopyIn(connection)
CS_CONNECTION   *connection;
{
    CS_BLKDESC *blkdesc;
    CS_DATAFMT datafmt;/* variable descriptions */
    Blk_Data *dptr;/* data for transfer */
    CS_INTdatalen[5];/* variable data length */
    CS_INT len;
    CS_INT numrows;


   /*
    ** Ready to start the bulk copy in now that all the
    ** connections have been made and have a table name.
    ** Start by getting the bulk descriptor initializing.
    */
    ...CODE DELETED.....

   /*
    ** Now to bind the variables to the columns and
    ** transfer the data
    */
    datafmt.locale = 0;
    datafmt.count = 1;
    dptr = BLKDATA;
    while (dptr->pub_id != DATA_END)
    {
        datafmt.datatype = CS_INT_TYPE;
        datafmt.maxlength = sizeof(CS_INT);
        datalen[0] = CS_UNUSED;

       if (blk_bind(blkdesc, 1, &datafmt, &dptr->pub_id,
            &datalen[0], NULL) != CS_SUCCEED)
        {
            ex_error("BulkCopyIn: blk_bind(1) failed");
            return CS_FAIL;
        }
        datafmt.datatype = CS_CHAR_TYPE;
        datafmt.maxlength = MAX_PUBNAME - 1;
        datalen[1] = strlen(dptr->pub_name);
        if (blk_bind(blkdesc, 2, &datafmt, dptr->pub_name,
             &datalen[1], NULL) != CS_SUCCEED)
        {
            ex_error("BulkCopyIn: blk_bind(2) failed");
            return CS_FAIL;
        }
```

```
            datafmt.maxlength = MAX_PUBCITY - 1;
            datalen[2] = strlen(dptr->pub_city);
            if (blk_bind(blkdesc, 3, &datafmt, dptr->pub_city,
                 &datalen[2], NULL) != CS_SUCCEED)
            {
                ex_error("BulkCopyIn: blk_bind(3) failed");
                return CS_FAIL;
            }
            datafmt.maxlength = MAX_PUBST - 1;
            datalen[3] = strlen(dptr->pub_st);
            if (blk_bind(blkdesc, 4, &datafmt, dptr->pub_st,
                 &datalen[3], NULL) != CS_SUCCEED)
            {
                ex_error("BulkCopyIn: blk_bind(4) failed");
                return CS_FAIL;
            }
            datafmt.maxlength = MAX_BIO - 1;
            datalen[4] = strlen((char *)dptr->pub_bio);
            if (blk_bind(blkdesc, 5, &datafmt, dptr->pub_bio,
                 &datalen[4], NULL) != CS_SUCCEED)
            {
                ex_error("BulkCopyIn: blk_bind(5) failed");
                return CS_FAIL;
            }
            if (blk_rowxfer (blkdesc) == CS_FAIL)
            {
                ex_error("BulkCopyIn: blk_rowxfer() failed");
                return CS_FAIL;
            }
            dptr++;
        }

        /* ALL the rows sent so clear up */
        ...CODE DELETED.....

        return CS_SUCCEED;
    }
```

Usage

- blk_rowxfer is a client-side routine.
- blk_rowxfer is equivalent to calling blk_rowxfer_mult with a NULL *row_count* parameter.
- See blk_rowxfer_mult in this chapter.

See also

blk_bind, blk_rowxfer_mult, blk_textxfer

blk_rowxfer_mult

Description

Transfers one or more rows during a bulk-copy operation.

Syntax

```
CS_RETCODE blk_rowxfer_mult(blkdesc, row_count)

CS_BLKDESC      *blkdesc;
CS_INT          *row_count;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

row_count

A pointer to a CS_INT variable or NULL.

For bulk-copy-out operations, blk_rowxfer_mult returns with **row_count* set to the number of rows read by the call. If *row_count* is NULL, this information is not available to the application. (The application can call blk_done to determine how many rows have been transferred by the cumulative number of blk_rowxfer_mult calls since the last blk_done call—but it is simpler to use a row count variable.

For bulk-copy-in operations, blk_rowxfer_mult sends the number of rows specified by **row_count* to the server. If *row_count* is NULL or **row_count* is 0, then the number of rows specified by *datafmt->count* in previous calls to blk_bind are sent to the server.

row_count is used by applications that perform array binding. See “Array binding” on page 131.

Return value

blk_rowxfer_mult returns:

Table 4-8: blk_rowxfer_mult return values

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.
CS_BLK_HAS_TEXT	The row contains one or more columns which have been marked for transfer using blk_textxfer. The application must call blk_textxfer to transfer data for these columns row before calling blk_rowxfer_mult to transfer the next row.
CS_END_DATA	When copying data out from a database, blk_rowxfer_mult returns CS_END_DATA to indicate that all rows have been transferred. When copying data into a database, blk_rowxfer_mult does not return CS_END_DATA.
CS_ROW_FAIL	A recoverable error occurred while fetching a row. Applies to bulk-copy-out operations only. blk_rowxfer_mult sets **row_count* to indicate the number of rows transferred (including the row containing the error) and transfers no rows after that row. The next call to blk_rowxfer_mult will read rows starting with the row after the one where the error occurred. Recoverable errors include memory allocation failures and conversion errors (such as overflowing the destination buffer) that occur while copying row values to program variables. In the case of buffer-overflow errors, blk_rowxfer_mult sets the corresponding **indicator* variable(s) to a value greater than 0. Indicator variables must have been specified in the application's calls to blk_bind. When blk_rowxfer_mult returns CS_ROW_FAIL, the application must continue calling blk_rowxfer_mult to keep retrieving rows, or it can call ct_cancel to cancel the remaining results.

A common reason for a blk_rowxfer_mult failure is conversion error.

Usage

- blk_rowxfer_mult is a client-side routine.
- An application calls blk_rowxfer_mult to transfer rows between program variables (bound with blk_bind) and the database table:

  - During a bulk-copy-in operation, blk_rowxfer_mult copies data from program variables to the database.
  - During a bulk-copy-out operation, blk_rowxfer_mult copies data from the database and places it in program variables.
- Application variables must first be bound to table columns with blk_bind for blk_rowxfer_mult to read or write their contents.

blk_rowxfer_mult and bulk-copy-in operations

- To transfer rows into a database, an application calls blk_rowxfer_mult repeatedly to transfer values from program variables to the database table. See "Program structure for bulk-copy-in operations" on page 107 for the sequence of Bulk-Library calls used to transfer data into a database table.
- During bulk-copy-in operations, the value of blk_rowxfer_mult's **row_count* parameter overrides the array lengths that were passed to blk_bind (as *datafmt->count*). The number of rows transferred per call is determined as follows:
  - If the application passes the address of a row count variable as the *row_count* parameter, then blk_rowxfer_mult transfers either *datafmt->count* or **row_count* rows, whichever is smaller.
  - If the application passes *row_count* as NULL, blk_rowxfer_mult always transfers *datafmt->count* rows.

  For example, if an application was uploading 103 rows and it used array binding to transfer 10 rows at a time, the application would:

  - Pass *datafmt->count* as 10 in all calls to blk_bind
  - Set **row_count* to 10 for the first 10 calls to blk_rowxfer_mult
  - Set **row_count* to 3 for the final call to blk_rowxfer_mult
- To upload row data that contains large text or image column values, you can forgo array binding and use blk_textxfer together with blk_rowxfer_mult to send large values one piece at a time. See "Transferring large text or image values in chunks" on page 163 for details.
- A bulk-copy-in operation is not automatically terminated if blk_rowxfer_mult returns CS_FAIL. An application can continue to call blk_rowxfer_mult after correcting or discarding the problem row.

blk_rowxfer_mult and bulk-copy-out operations

- To transfer rows out of a database, an application calls blk_rowxfer_mult repeatedly to read column values from the server and place them in program variables. See "Program structure for bulk-copy-out operations" on page 109 for the sequence of Bulk-Library calls used to read data from a database table.
- For bulk copies out of a database, the use of blk_rowxfer_mult is similar to the use of the Client-Library ct_fetch routine.
- The number of rows to be read by blk_rowxfer_mult is determined by the value passed as *datafmt->count* in the application's calls to blk_bind. blk_rowxfer_mult attempts to read this number of rows and write the data to program variables.

  Fewer rows may be read by the final call to blk_rowxfer_mult (that is, the call that retrieves the last row in the table) or if a conversion error occurs while data is being retrieved. The former condition is indicated by a return code of CS_END_DATA; the latter, by CS_ROW_FAIL. In either case, blk_rowxfer_mult returns with **row_count* set to the actual number of rows read.
- To download row data that contains large text or image column values, you can forgo array binding and use blk_textxfer together with blk_rowxfer_mult to read large values one piece at a time. See the following section for details.

Transferring large *text* or *image* values in chunks

- If array binding is not in effect, an application can use blk_textxfer in conjunction with blk_rowxfer_mult to transfer rows containing large text or image values. For information on how to do this, see "Bulk-Library client programming" on page 103.
- For tables that contain large text or image columns, it is often convenient for an application to transfer the text or image data in fixed-size chunks rather than all at once. If a column is transferred all at once, the application must have sufficient buffer space to hold the value in its entirety.
- To transfer large column values in chunks:
  - The application passes *buffer* as NULL in its blk_bind call for the column. This setting specifies that data for this column will be transferred using blk_textxfer. For a bulk-copy-in operation, the application must also specify the length of the column value as blk_bind's **datalen* parameter.

- The application calls blk_rowxfer_mult to transfer the row. blk_rowxfer_mult returns CS_BLK_HAS_TEXT, indicating that Bulk-Library expects further data for this row to be transferred with blk_textxfer.
- For each column requiring transfer, the application calls blk_textxfer in a loop until blk_textxfer returns CS_END_DATA, indicating that all of the data for this column has been transferred.

See also

blk_bind, blk_textxfer

blk_sendrow

Description

A server-side routine, sends a formatted bulk-copy row obtained from *blk_getrow*.

Syntax

CS_RETCODE blk_sendrow(blkdesc, row)

```
CS_BLKDESC     *blkdesc;
CS_BLK_ROW     *row;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

row

A pointer to a CS_BLK_ROW structure. The CS_BLK_ROW is a hidden structure that holds formatted bulk-copy rows sent from the client. A gateway application can fill in a CS_BLK_ROW structure with a formatted row by calling the server-side routine blk_getrow.

Return value

blk_sendrow returns:

Table 4-9: blk_sendrow return values

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BLK_HAS_TEXT	The row contains one or more text, image, sensitivity, or boundary columns. The application must call blk_gettext and blk_sendtext to transfer the columns for this row before calling blk_getrow and blk_sendrow to transfer the next row.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.

Usage

- blk_sendrow is a server-side routine.
- A gateway application uses blk_sendrow in conjunction with blk_getrow. Together, the two routines enable a gateway application to receive formatted bulk-copy rows from an Open Client application and send them on to Adaptive Server Enterprise.
- blk_sendrow is a gateway-specific substitute for blk_rowxfer or blk_rowxfer_mult. An application can call blk_sendrow only after calling blk_getrow to retrieve a formatted row.
- The sequence of calls in the gateway application is:
  - blk_getrow, to obtain a formatted bulk-copy row
  - blk_sendrow, to send the formatted row to Adaptive Server Enterprise

  If blk_getrow returns CS_BLK_HAS_TEXT, the application must call the following routines in a loop, until blk_gettext returns CS_END_DATA:
  - blk_gettext, to pick up a chunk of text, image, sensitivity, or boundary data
  - blk_sendtext, to send a chunk of text, image, sensitivity, or boundary data

  Only one blk_gettext/blk_sendtext loop is required, no matter how many text, image, sensitivity, or boundary columns are being transferred.

See also

blk_init, blk_sendtext, blk_colval, blk_getrow, blk_gettext

blk_sendtext

Description

A server-side routine, sends text, image, sensitivity, or boundary data in a formatted bulk-copy row obtained from blk_getrow.

Syntax

```
CS_RETCODE blk_sendtext(blkdesc, row, buffer,
        buflen)

CS_BLKDESC    *blkdesc;
CS_BLK_ROW    *row;
CS_BYTE       *buffer;
CS_INT        buflen;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

row

A pointer to a CS_BLK_ROW structure. The CS_BLK_ROW structure is a hidden structure that holds formatted bulk-copy rows sent from the client. A gateway application can fill in a CS_BLK_ROW structure with a formatted row by calling the blk_getrow routine.

buffer

A pointer to the space from which blk_sendtext picks up the chunk of text, image, sensitivity, or boundary data.

buflen

The length, in bytes, of the **buffer* data space.

Return value

blk_sendtext returns:

Table 4-10: blk_sendtext return values

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.

Usage

- blk_sendtext is a client-side routine.
- A gateway application uses blk_sendtext in conjunction with blk_gettext. Together, the two routines enable a gateway application to receive chunks of text, image, sensitivity, or boundary data in formatted bulk-copy rows from an Open Client application and send them on to Adaptive Server Enterprise.

- blk_sendtext is a gateway-specific substitute for blk_textxfer. An application can call blk_sendtext only after calling blk_gettext to retrieve a chunk of text, image, sensitivity, or boundary data belonging to a formatted row.
- The sequence of calls in the gateway application is:
  - blk_getrow, to pick up a formatted bulk-copy row
  - blk_sendrow, to send the formatted row to Adaptive Server Enterprise

  If blk_sendrow returns CS_BLK_HAS_TEXT, the application must call the following routines in a loop, until blk_gettext returns CS_END_DATA:
  - blk_gettext, to pick up a chunk of text, image, sensitivity, or boundary data
  - blk_sendtext, to send a chunk of text, image, sensitivity, or boundary data

  Only one blk_gettext/blk_sendtext loop is required, no matter how many text, image, sensitivity, or boundary columns are being transferred.

See also

blk_init, blk_sendrow, blk_colval, blk_getrow, blk_gettext

blk_srvinit

Description

A server-side routine, copies descriptions of server table columns to the client, if required.

Syntax

CS_RETCODE blk_srvinit(srvproc, blkdescp)

```
SRV_PROC      *srvproc;
CS_BLKDESC    *blkdescp;
```

Parameters

srvproc

A pointer to the SRV_PROC structure associated with the client receiving column descriptions. It contains all the information that Server-Library uses to manage communications and data between the Open Server application and the client.

blkdescp
A pointer to a structure containing information about bulk-copy data. This structure must have been previously allocated with a call to blk_alloc and initialized through a call to blk_init. This structure is used to correctly interpret incoming formatted bulk-copy rows.

Return value

blk_srvinit returns:

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed; no action was taken.

Usage

- blk_srvinit is a server-side routine that is useful in gateway applications.
- This routine sends the current server table column descriptions in the CS_BLKDESC structure to the client, if the client's TDS version is 5.0 or later.
- This routine must be called from within a SRV_LANGUAGE event handler, in response to an "insert bulk" request from the client.
- Once blk_srvinit has successfully returned descriptions to the client, the Open Server application's SRV_BULK event handler can begin reading bulk data from the client. The event handler first calls blk_rowalloc, then calls blk_getrow and blk_sendrow in a loop to transfer the bulk-copy rows.
- blk_init places the descriptions in the CS_BLKDESC structure, so the gateway application must call blk_init before calling blk_srvinit.

See also

blk_init, blk_getrow, blk_rowalloc, blk_sendrow

blk_textxfer

Description

Transfers a column's data in chunks during a bulk-copy operation.

Syntax

```
CS_RETCODE blk_textxfer(blkdesc, buffer, buflen,
        outlen)

CS_BLKDESC    *blkdesc;
CS_BYTE       *buffer;
CS_INT         buflen;
CS_INT        *outlen;
```

Parameters

blkdesc

A pointer to the CS_BLKDESC that is serving as a control block for the bulk-copy operation. blk_alloc allocates a CS_BLKDESC structure.

buffer

A pointer to the space from which blk_textxfer picks up the chunk of text, image, sensitivity, or boundary data.

buflen

The length, in bytes, of the **buffer* data space.

outlen

A pointer to an integer variable. *outlen* is not used for a bulk-copy-in operation and should be passed as NULL.

For a bulk-copy-out operation, **outlen* represents the length, in bytes, of the data copied to **buffer*.

Return value

blk_textxfer returns:

Table 4-11: blk_textxfer return values

Returns	Indicates
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_END_DATA	When copying data out from a database, blk_textxfer returns CS_END_DATA to indicate that a complete column value has been sent. When copying data into a database, blk_textxfer returns CS_END_DATA when an amount of data equal to blk_bind's **datalen* has been sent.
CS_PENDING	Asynchronous network I/O is in effect. See the "Asynchronous Programming" topics page in the *Open Client Client-Library/C Reference Manual*.

Examples

```
/*
  ** BulkCopyIn()
  **
  ** BLKDATA and DATA_END are defined in the bulk copy
  ** example program.
  */

  CS_STATIC CS_RETCODE
  BulkCopyIn(connection)
  CS_CONNECTION   *connection;
  {
```

```
CS_BLKDESC   *blkdesc;
CS_DATAFMT   datafmt;      /* variable descriptions */
Blk_Data     *dptr;        /* data for transfer */
CS_INT       datalen[5];   /* variable data length */
CS_INT       len;
CS_INT       numrows;

/*
** Ready to start the bulk copy in now that all the
** connections have been made and have a table name.
** Start by getting the bulk descriptor initializing.
*/
...CODE DELETED.....

/* Bind columns and transfer rows */
dptr = BLKDATA;
while (dptr->pub_id != DATA_END)
{
     datafmt.datatype = CS_INT_TYPE;
     datafmt.count = 1;
     datafmt.maxlength = sizeof(CS_INT);
     datalen[0] = CS_UNUSED;

     if (blk_bind(blkdesc, 1, &datafmt, &dptr->pub_id,
          &datalen[0], NULL) != CS_SUCCEED)
     {
          ex_error("BulkCopyIn: blk_bind(1) failed");
          return CS_FAIL;
     }
     datafmt.datatype = CS_CHAR_TYPE;
     datafmt.maxlength = MAX_PUBNAME - 1;
     datalen[1] = strlen(dptr->pub_name);
     if (blk_bind(blkdesc, 2, &datafmt, dptr->pub_name,
           &datalen[1], NULL) != CS_SUCCEED)
     {
          ex_error("BulkCopyIn: blk_bind(2) failed");
          return CS_FAIL;
     }
     datafmt.maxlength = MAX_PUBCITY - 1;
     datalen[2] = strlen(dptr->pub_city);
     if (blk_bind(blkdesc, 3, &datafmt, dptr->pub_city,
           &datalen[2], NULL) != CS_SUCCEED)
     {
          ex_error("BulkCopyIn: blk_bind(3) failed");
          return CS_FAIL;
     }
```

```
                    datafmt.maxlength = MAX_PUBST - 1;
                    datalen[3] = strlen(dptr->pub_st);
                    if (blk_bind(blkdesc, 4, &datafmt, dptr->pub_st,
                          &datalen[3], NULL) != CS_SUCCEED)
                    {
                         ex_error("BulkCopyIn: blk_bind(4) failed");
                         return CS_FAIL;
                    }
                    datafmt.datatype = CS_TEXT_TYPE;
                    datafmt.maxlength = MAX_BIO - 1;
                    datalen[4] = strlen((char *)dptr->pub_bio);
                    if (blk_bind(blkdesc, 5, &datafmt, NULL,
                          &datalen[4], NULL) != CS_SUCCEED)
                    {
                         ex_error("BulkCopyIn: blk_bind(5) failed");
                         return CS_FAIL;
                    }
                    if (blk_rowxfer (blkdesc) == CS_FAIL)
                    {
                         ex_error("BulkCopyIn: EX_BLK - Failed on \
                              blk_rowxfer.");
                         return CS_FAIL;
                    }
                    if (blk_textxfer(blkdesc, dptr->pub_bio,
                         datalen[4], &len) == CS_FAIL)
                    {
                         ex_error("BulkCopyIn: blk_rowxfer() failed");
                         return CS_FAIL;
                    }
                    dptr++;
               }
               /* ALL the rows sent so clear up */
               ...CODE DELETED.....

               return CS_SUCCEED;
          }
```

Usage

- blk_textxfer is a client-side routine.
- blk_textxfer transfers large text or image values. blk_textxfer does not perform any data conversion; it simply transfers data.
- There are two ways for an application to transfer text and image values during a bulk-copy operation:

  - The application can treat text or image data like ordinary data: that is, it can bind columns to program variables and transfer rows using blk_rowxfer_mult. Generally, this method is convenient for small text and image values but not for larger ones. If the entire value is to be transferred by blk_rowxfer_mult, the application must allocate program variables that are large enough to hold entire column values.
  - Using blk_textxfer, the application can transfer text or image data in chunks. This method allows the application to use a transfer buffer that is smaller than the values to be transferred.

- An application marks a column for transfer through blk_textxfer by calling blk_bind for the column with a NULL *buffer* parameter. If the transfer is going into the database, pass the total length of the value as blk_bind's **datalen* parameter.
- See Chapter 3, "Bulk-Library."

Using *blk_textxfer* for bulk-copy-in operations

- An application's blk_bind calls do not have to be in column order, but data for blk_textxfer columns must be transferred in column order.

  For example, an application can bind columns 3 and 4, and then mark columns 2 and 1 for transfer using blk_textxfer. After calling blk_rowxfer_mult to copy data for columns 3 and 4, the application needs to call blk_textxfer to transfer data for column 1 before calling it for column 2.
- When copying data into a database, if a text, image, boundary, or sensitivity datatype column is marked for transfer using blk_textxfer, all subsequent columns of these types must also be marked for transfer using blk_textxfer.

  For example, an application cannot mark the first text column in a row for transfer using blk_textxfer and then bind a subsequent text column to a program variable.
- When copying data into a database, an application is responsible for calling blk_textxfer the correct number of times to transfer the complete text or image value.

Using *blk_textxfer* for Bulk-Copy-Out operations

- When using blk_textxfer to copy data out of a database, only columns that follow bound columns are available for transfer using blk_textxfer. In other words, columns being transferred using blk_textxfer must reside at the end of row.

For example, an application cannot bind the first two columns in a row to program variables, mark the third for transfer using blk_textxfer, and bind the fourth.

- When copying data out from a database, blk_textxfer returns CS_END_DATA to indicate that a complete column value has been copied.

See also blk_bind, blk_rowxfer_mult

Index

A

B

C

D

E

G

H

I

L

M

N

O

P

R

S

T

U

V